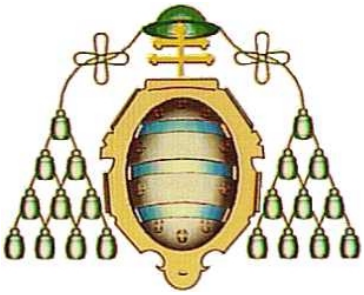
**UNIVERSIDAD DE OVIEDO**

Departamento de Informática



TESIS DOCTORAL

# Sistema de verificación de componentes software

Presentada por

Agustín Cernuda del Río

para la obtención del título de Doctor en Informática

Dirigida por el

Profesor Doctor D. Juan Manuel Cueva Lovelle

Oviedo, Febrero de 2002

# Resumen

Esta tesis describe un sistema de verificación de componentes software cuyo fin es detectar a priori las conexiones no válidas que se producen al combinar los componentes. Esta verificación se realiza además de manera estática, automática, asequible y basada en el conocimiento disponible sobre los componentes.

El uso de componentes software que se combinan para construir aplicaciones es una tendencia creciente en la industria del desarrollo. Del uso de estos componentes se espera reducir costes de desarrollo, acortar el tiempo de salida al mercado de los productos software y mejorar la calidad de los mismos.

La motivación para esta tesis surge de experiencias profesionales en las que quedó patente que modelos de componentes como COM, CORBA o JavaBeans resuelven problemas de interoperabilidad a nivel de implementación pero no ofrecen suficiente protección al programador para evitar la violación de las restricciones de funcionamiento de cada componente; se limitan a verificar la compatibilidad entre signaturas. Pueden producirse multitud de problemas que no son originados por mal funcionamiento de un componente particular, sino por una combinación incorrecta (en un sentido funcional) de varios de ellos.

Muchos de estos problemas podrían evitarse si se hubiera tenido en cuenta todo el conocimiento que a priori se tiene sobre cada componente. Estos conocimientos, estas "instrucciones de funcionamiento", frecuentemente toman la forma de un documento en lenguaje natural, que el programador debe leer y aplicar, con la consiguiente posibilidad de cometer errores. En algunos casos, las especificaciones de funcionamiento toman la forma de precondiciones y postcondiciones, pero esto suele resultar aplicable sólo en tiempo de ejecución, una vez que el software se ha construido.

Existen diversas técnicas que podrían aplicarse a esta verificación, pero en general adolecen de alguno de estos problemas: ser demasiado complejas o formales, difíciles de entender y/o aplicar, poco adecuadas para el trabajo con componentes, requerir avances teóricos antes de ser útiles en la práctica, ser específicas y no de uso general a diversos niveles, o ser rechazadas por la industria por alguna combinación de estas razones.

En esta tesis se propone, pues, una solución al problema planteado que tiene en cuenta estas restricciones de viabilidad inmediata y transferencia tecnológica, así como las restricciones técnicas (verificación estática y automática) enunciadas anteriormente.

**Palabras clave**: Componentes software, modelo de componentes, ingeniería del software, verificación, transferencia tecnológica, programación lógica con restricciones, gestión del conocimiento, verificación estática.

# Abstract

This dissertation describes a software components verification system whose aim is the *a priori* detection of the invalid connections which arise when combining components. In addition, this verification is done in a static, automatic and viable way, and is based on the available knowledge about the involved components.

The use of software components which are combined in order to build applications is a growing trend in the software development industry. The use of these components is expected to bring a development cost reduction, a shorter time-to-market and a better quality of the final product.

This thesis was motivated by professional experiences in which it became clear that component models such as COM, CORBA or JavaBeans solve interoperability problems at the implementation level, but they do not protect the programmer sufficiently against violating the working restrictions of each component; they simply verify signature compatibility. Many problems can arise that are not caused by a malfunction in any particular component, but by an incorrect (in a functional sense) combination of several components.

Many of these problems could be avoided if all available knowledge about each component was taken into account. This knowledge, this "operation manual", is frequently in the form of a natural language document to be read and applied by the programmer, with the consequent possibility of making mistakes. In some cases, working specifications are stated as preconditions and postconditions, but this is usually useful only at run time, once the software has been built.

There are several techniques that could be applied to this verification process, but in general terms they suffer from some of the following problems: they are too complex or formal, or difficult to understand and/or use, or inadequate for working with components, or require theoretical advances before being useful in practice, or are specific and not for general use at different levels, or are rejected by the industry for some combination of these reasons.

In this dissertation we propose a solution to the described problem that takes into account these issues of immediate viability and technology transfer, and also the technical restrictions (static, automatic verification) previously stated.

**Keywords**: Software components, component model, software engineering, verification, technology transfer, constraint logic programming, knowledge management, static verification.

# Agradecimientos

A mi madre, a quien debo todo.
A la memoria de mi padre.

Muchas personas han intervenido de diversas formas para que yo haya podido llegar a este punto. Mi director de tesis, Juan Manuel Cueva, y mi amigo y compañero Jose Emilio Labra me han enseñado en qué consistía todo esto y me han ayudado desde que empecé con la tesis hace ya varios años. Mis compañeros de Seresco me enseñaron con los hechos gran parte de lo que sé sobre desarrollo de software y calidad en el mundo real, pero también sobre ética de trabajo. A otros investigadores debo agradecer su ayuda, su paciencia, su generosidad y su inspiración; es el caso de Carine Lucas, Kurt Wallnau, Dick Hamlet, François Thomasset, Denise Woit, Wolfgang Weck, Dirk Deridder, John W. Chinneck, y tantos y tantos otros. Y a mis compañeros actuales en la Universidad de Oviedo, con quienes he resuelto el tramo final de este trabajo durante el último año, les debo un entorno lleno de generosidad y de talento, que llega mucho más allá del esfuerzo que absorbe el trabajo diario en condiciones muy difíciles y sin ayuda de nadie. Mencionaré sólo a Daniel Gayo, con quien he trabajado estrechamente en muchas cosas, y que hace muchos meses apostó a que tendría listo el primer borrador en Diciembre de 2001 (y acertó); pero la lista sería larga, porque esto está plagado de gente excepcional.

Sin duda, en este capítulo merece un apartado propio Elena López, que no sólo me ha ayudado de manera directa en el plano técnico, sino que ha derrochado esfuerzo de otras mil maneras. Sería absurdo –e inútil- intentar enumerarlas aquí; sólo puedo dejar constancia de que entre estos párrafos hay probablemente tanto trabajo suyo como mío. Darle las gracias es sólo una pobre forma de hacer justicia.

# Tabla de contenidos

# 1. Introducción

## 1.1. Formulación básica del problema

La construcción de software está considerada como uno de los problemas técnicos cruciales de nuestro tiempo; desde la aparición del software a mediados del siglo XX, este ha ido ganando complejidad. En un período de tiempo relativamente corto, el software pasó de ser un elemento considerado secundario (frente al enorme coste de los dispositivos físicos involucrados en la fabricación de los primeros ordenadores) a absorber una cantidad cada vez mayor de recursos económicos y acabar convirtiéndose en un factor decisivo de cara al éxito o fracaso de un sistema basado en computadora. Resulta caro desarrollar software, y los defectos del software son particularmente costosos, tanto por los gastos de mantenimiento que generan como por las repercusiones en la imagen del producto y la satisfacción de los usuarios. Hoy en día, existe una clara conciencia sobre la importancia de la calidad del software.

Dentro de esta búsqueda de una mayor calidad y un menor coste, ha ido surgiendo una tendencia hacia el uso de componentes software. La idea subyacente a esta tendencia es que, mientras que la construcción de software a medida es un proceso caro y complejo, buena parte de la funcionalidad de un sistema es lo suficientemente "genérica" como para que se adquiera de terceras partes bajo la forma de los así llamados *componentes software*: porciones ejecutables de distribución independiente, con una interfaz claramente definida, preparadas para ser utilizadas por terceros, que no tienen un estado persistente y que interactúan para formar un sistema funcional [Sz97, p. 30]. Esto traería consigo un ahorro de tiempo (al evitar fases completas de desarrollo) y un aumento de fiabilidad (puesto que los componentes disponibles comercialmente se suponen probados y validados por el mercado). Más aún, es posible diseñar el sistema de forma que los elementos que necesariamente hay que desarrollar a medida pueden también adoptar la forma de componentes, facilitando su sustitución y la creación de familias de productos a partir de una estructura común.

La idea general de construir sistemas de software mediante la conjunción de componentes se aplica, de hecho, aun cuando no interviene la compra de estos elementos a terceros; los propios paradigmas de la orientación a objetos o de la programación modular recogen algunas de estas ideas (si bien el término *componente*, en general, se reserva para la acepción descrita más arriba). Si se dejan aparte las consideraciones sobre distribución y empaquetado de cara a la venta, básicamente un componente sería un fragmento de software, con una interfaz claramente delimitada, y cuyos detalles internos resultan indiferentes –y de hecho desconocidos- para su usuario. Este *encapsulamiento* trae como

consecuencia una disminución de complejidad que constituye otro de los beneficios esenciales del uso de componentes.

No obstante, existen problemas ligados a la construcción de un sistema a partir de componentes (ya se use el término *componente* en su acepción más extendida o en un sentido más flexible). Muchos de los fallos de un sistema de software no tienen su origen en el seno de un componente individual, sino en la interconexión de este componente con otros, y ello se debe a que de alguna forma se violan las condiciones de funcionamiento correcto de este componente. En efecto, un componente presenta unos puntos de conexión con el exterior, interactúa con su entorno, y en esa interacción espera que se cumplan ciertas condiciones. Si esto no es así, se produce un fallo del sistema que en cierta medida cabe calificar de *comportamiento emergente*. A pesar de esto, los métodos actuales de descripción de interfaces suelen centrarse en aspectos puramente sintácticos y estructurales, obviando las restricciones de comportamiento [KJB00, p. 4; CFPTV01, p. 1].

Lo que aquí se pretende desvelar es si existe un método que permita verificar que cierta combinación de componentes satisface los requisitos planteados por cada uno de ellos. Dicho de otro modo, dada una combinación de componentes se pretende señalar en qué *conexiones* no se están garantizando las condiciones de funcionamiento adecuadas para los componentes involucrados. Se pretende realizar la verificación con el grado de confianza que pueda proporcionar el conocimiento disponible sobre los componentes involucrados; básicamente, las interfaces sintácticas se acompañarían con información sobre restricciones de funcionamiento.

Sorprendentemente, esta no es una práctica extendida en la industria. Se tendrá ocasión de comprobar que en general los componentes software llevan implícita una descripción de interfaces, pero limitada a las posibles operaciones o servicios que ofrecen y al número y tipo de parámetros involucrados (lo que se denomina *signatura*). El contacto mantenido con el mundo profesional del desarrollo de software corrobora esta tesis. Por poner un ejemplo, la obra básica de referencia en componentes software, [Sz97], apenas menciona la verificación de los problemas que aquí estamos describiendo.

Es nuestra opinión que resulta posible dotar a los componentes de especificaciones que vayan mucho más allá de las signaturas de las operaciones, y esto resultaría además muy ventajoso de cara al desarrollo con componentes. Además, el paradigma de componentes puede ser utilizado para verificar aspectos del desarrollo mucho más variados que la combinación de módulos binarios y ejecutables comprados a terceros. De ahí la tesis que aquí se plantea, cuya formulación resumida podría ser la frase siguiente:

> *Es posible verificar, de manera estática, automática y asequible que, hasta donde nos es posible asegurar con el conocimiento disponible, al combinar ciertos componentes software no se han violado las condiciones de funcionamiento correcto de ninguno de ellos.*

Esta es una expresión en forma muy condensada, a efectos de centrar el ámbito del trabajo de investigación; muchos de los términos incluidos en ella tienen un significado implícito que no resulta obvio, y exigen un desarrollo adicional que se abordará en esta disertación. Por el momento, se ofrecerá una breve expansión de los mismos, y más adelante se entrará en una discusión más detallada.

Se desea realizar la verificación de manera **estática**. Esto quiere decir que se pretende realizar la verificación sin llegar a poner el sistema en funcionamiento (incluso posiblemente sin construirlo): a priori, basándose en un análisis del conocimiento que se tiene sobre los componentes.

La verificación debe ser además **automática**, esto es, ser realizada por un ordenador y no por un analista humano. Esto redunda también en que el resultado de la verificación sea no ambiguo.

También se afirma que el proceso de verificación debe ser **asequible**. Con el término "asequible" englobamos aquí varias restricciones:

- Susceptible de ser implementado mediante técnicas conocidas y viables.
- Susceptible de ser comprendido y aplicado en la práctica con relativa facilidad por personal de desarrollo.
- Flexible y susceptible de ser aplicado en diferentes ámbitos, no demasiado específico.

La expresión **con el conocimiento disponible** indica que se debe ofrecer alguna vía para que el conocimiento que los desarrolladores tienen sobre los requisitos de los componentes quede plasmado y sea utilizado en su totalidad. Asimismo, no se pretende un método de verificación exhaustiva por observación directa del código fuente, sino precisamente un método que aproveche ese conocimiento que se tiene a priori.

El resto de esta disertación pretende apoyar esta tesis. Siendo su ámbito la ingeniería del software, y siendo este un campo en el que en muchos casos no resulta fácil o incluso viable realizar mediciones o demostraciones matemáticas, tal apoyo se ofrece por dos vías: una constructiva, proponiendo un método que responde a los requisitos planteados, y otra experimental, poniendo a prueba dicho método en diferentes casos de estudio.

## 1.2. Interés del problema

Responder a la cuestión planteada en la formulación del problema merece la pena por varias razones. A continuación se evaluarán los diferentes términos en que se ha planteado el problema, justificando someramente su presencia en el enunciado. Por supuesto, a lo largo de esta disertación se realizará esta misma tarea de forma más detallada y razonada.

Se pretende detectar los problemas que surgen al combinar componentes software. Por razones ya mencionadas y que pueden ampliarse en [Sz97], la construcción de programas a partir de componentes es una tendencia en alza. Teniendo en cuenta que muchos defectos no son achacables a componentes concretos, sino a incoherencias que surgen de su combinación [GAO95, GB98], parece necesario desarrollar técnicas específicas para su prevención.

Esa verificación debe ser además estática. El interés de una verificación estática radica en primer lugar en que la prueba del software plantea considerables dificultades; generalmente, la prueba exhaustiva es inviable debido al enorme tamaño potencial del espacio de estados de un programa, lo que conlleva que se prueben sólo estados "significativos" según ciertos heurísticos. Parece razonable pensar que si se aprovecha todo el conocimiento que a priori se tiene sobre los componentes, y se obtienen conclusiones analíticas a partir del mismo, se

podrán detectar defectos que de este modo no pasarán a fases posteriores (incluida la fase de pruebas). La verificación estática, además, tiene la ventaja de que puede realizarse sin ejecutar el software; en este caso, el modelo propuesto puede aplicarse incluso sin construirlo, en tiempo de diseño.

El interés de una verificación automática proviene del hecho de que esta verificación previsiblemente se realizará con menor coste y más frecuencia y fiabilidad, al igual que ocurre con las pruebas de regresión automatizadas frente a las pruebas manuales. Además, una verificación automática tiene como ventaja que por lo general no es ambigua; la verificación manual puede producir buenos resultados [NASA93], pero es también más cara y subjetiva.

Un aspecto muy importante de la tesis es que el proceso de verificación sea asequible. Este término es sólo un resumen, y engloba varios aspectos, como se detalló en 1.1. El uso de técnicas de verificación y validación no parece estar lo suficientemente extendido en la industria; gran parte del interés de esta tesis reside en los estrictos condicionantes de viabilidad práctica de la solución aportada, ya que en muchas otras propuestas tal viabilidad se pone frecuentemente en entredicho, ya sea técnicamente o respecto a recursos humanos (y a efectos de adopción por la industria es indiferente que tales reservas sean justificadas o sólo una percepción errónea). En palabras de Parnas [Ei99] ante la pregunta de cuáles eran las ideas o técnicas de ingeniería del software más prometedoras que había en el horizonte:

> *El mayor beneficio no provendrá de nuevas investigaciones sino de poner las viejas ideas en práctica y enseñar a la gente cómo aplicarlas correctamente. Hay mucha más investigación que hacer y tenemos mucho que aprender, pero la prioridad debería estar en la transferencia de tecnología y en la educación.*

El que el método se pueda implementar mediante técnicas conocidas y viables es una primera restricción; eliminaría la incertidumbre sobre la posibilidad de construir herramientas apropiadas. La posibilidad de que el personal de desarrollo típico domine el método es otra restricción cuya utilidad parece clara (menores costes de formación y adopción). Su flexibilidad, por último, redundaría en una mayor utilidad del método en ámbitos diferentes, y por tanto un mayor retorno de inversión.

El utilizar el conocimiento disponible tiene una doble vertiente. En primer lugar, parece útil ofrecer una vía para que los desarrolladores de componentes establezcan explícitamente todas las restricciones que el componente plantea y todas las garantías que ofrece. En segundo lugar, existen métodos que analizan el código fuente y detectan defectos; pero parece apropiado intentar llegar a conclusiones a partir del conocimiento que a priori se tiene sobre los componentes. El propósito funcional de un componente, por ejemplo, difícilmente puede deducirse de manera automática analizando el código fuente (recuérdense, además, las ventajas mencionadas del análisis estático cuando este no requiere disponer del código fuente). El aprovechar el conocimiento disponible permitiría incluir en la verificación criterios ajenos al código fuente, pero conocidos por los diseñadores y programadores.

## 1.3. Resultados

Como se mostrará a lo largo de esta tesis, se ha abordado el problema por el procedimiento de idear un modelo (denominado Itacio) que dé respuesta a las cuestiones planteadas, y luego preparar un prototipo que ayude a reflexionar sobre el modelo y refinarlo (repitiendo esta secuencia iterativamente). Durante todo el proceso, se ha tenido muy en cuenta el condicionante de simplicidad. Asimismo, se ha aplicado el modelo de verificación (de manera experimental) a ámbitos que no estaban previstos de antemano; esta improvisación prueba su versatilidad. Por otra parte, la preparación de los prototipos ha permitido evaluar la viabilidad técnica y la posibilidad de automatizar el proceso. Todos los prototipos se han preparado con tecnologías claramente asequibles y con recursos extremadamente limitados, lo que permite suponer que la creación de un sistema plenamente funcional es perfectamente posible.

El método propuesto pretende ser, pues, una forma de probar nuestra tesis. Además de ese fin instrumental, el propio método constituye una aportación a la Ingeniería del Software, puesto que entendemos que es aplicable en la práctica previa construcción de las herramientas adecuadas. En el proceso de verificación de esta tesis se ha llevado a cabo, además, un estudio de diversas técnicas relacionadas (siempre a la luz de los condicionantes establecidos por nuestra tesis) que a nuestro entender también constituye una aportación. Los usos experimentales de Itacio resultan también de interés porque en algunos casos representan la aplicación del paradigma de componentes a ámbitos que tradicionalmente no se han venido encuadrando en su campo de acción (y en especial, por supuesto, no se les ha venido aplicando el tipo de verificación que aquí se propone como principal novedad).

## 1.4. Organización de este documento

Esta disertación está organizada de la forma siguiente. Este primer capítulo representa una visión general de la disertación, presentando la tesis, justificando su interés y resumiendo de forma muy somera sus resultados.

El capítulo 2 representa un recorrido de las técnicas previamente existentes y que podrían relacionarse con los fines que aquí se persiguen. Algunas de estas técnicas presentan puntos en común con el modelo Itacio, aunque ninguna de ellas satisface plenamente nuestra tesis.

El capítulo 3 presenta con mayor profundidad el problema que se pretende resolver, explicando en detalle los términos de la tesis. Tras una descripción de la misma, se realiza un análisis detallado del trabajo existente en los diversos campos, análisis que para mayor comodidad del lector se estructura de forma análoga al estudio del capítulo 2. Durante este análisis se evalúa en qué medida las diversas soluciones y técnicas existentes responden o no satisfactoriamente a las exigencias planteadas por esta tesis. También como parte del capítulo 3 se justifica el interés de la tesis en los términos en los que está planteada.

El capítulo 4 presenta el modelo Itacio. Se describen sus elementos, la relación entre los mismos y los algoritmos de verificación involucrados. Se discuten algunos aspectos de implementación u otros temas relacionados con el modelo.

La definición del modelo es una parte importante de la solución aportada, pero no es el único aspecto de la misma; la aplicación de este modelo en diversos casos de estudio tiene un papel fundamental en su validación. En el capítulo 5 se presentan los diversos

experimentos realizados. Comienza con una descripción de los 3 diferentes prototipos de herramienta de verificación basada en Itacio que se han desarrollado para los experimentos; esto permite valorar la viabilidad de ese desarrollo y el grado de disponibilidad de las técnicas implicadas. Después de eso, se ofrece información detallada sobre algunos casos concretos de uso de esos prototipos, y las conclusiones obtenidas de esas experiencias. Al resultar muy difícil de realizar una validación cuantitativa de la idoneidad del modelo, otra vía que se ha utilizado para sopesar la corrección de la tesis es someter estas ideas a escrutinio público, en foros especializados en los diferentes aspectos de Itacio (desarrollo basado en componentes, ingeniería del software, programación lógica, gestión del conocimiento...) La parte final de este capítulo describe esta interacción con otros expertos y las conclusiones obtenidas.

Finalmente, se ofrecen otros apartados centrados en las conclusiones, la evaluación de futuras líneas de investigación y la recopilación de la bibliografía relacionada.

# 2. Técnicas relacionadas con la verificación de conglomerados de componentes

En esta sección se presentan las principales ideas y enfoques que pueden resultar de relevancia para el tema abordado en esta tesis. Posteriormente, en el capítulo 3, se discutirá en qué grado constituyen una respuesta suficiente a la misma; en dicho capítulo se ofrecen, como parte de ese análisis, algunos datos adicionales.

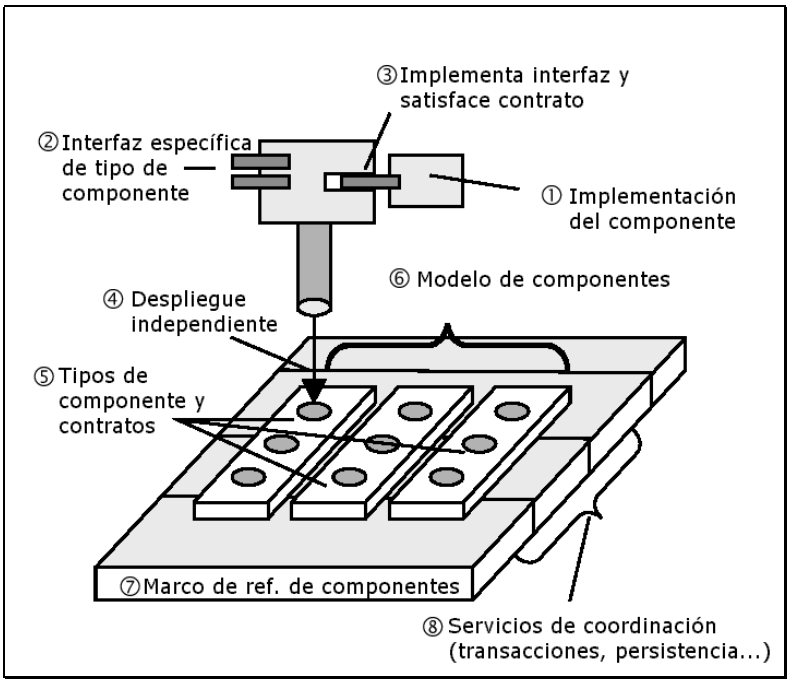



*Ilustración 1. Modelo de referencia para tecnología de componentes software (según [WP00])*

## 2.1. Definiciones previas

La propia definición de "componente" puede plantear un largo debate, puesto que no existe consenso al respecto [KJB00]. De hecho, esta misma disertación puede considerarse como un claro ejemplo de heterodoxia, ya que la noción de componente que aquí se propone es deliberadamente abierta y flexible.

No existiendo consenso, lo que sí existe es un punto de referencia común, una definición "clásica" de componente. Este punto de vista ha sido resumido convenientemente por Szyperski [Sz97] y la comunidad del desarrollo de software se adhiere mayoritariamente a sus definiciones. Respecto a la comunidad de investigadores, aunque no todos acepten ese mismo enfoque, sí que reconocen que existe una definición "clásica" o "más extendida" y que coincide con la propuesta por Szyperski. Así pues, en esta tesis se recurre también a este punto de referencia a partir del cual se pueda divergir más o menos, y cuando se mencione "la definición típica", "el concepto clásico" y expresiones similares, se estará mencionando implícitamente el siguiente conjunto de definiciones debidas a Szyperski:

**Def. I**: Un **componente software** es una unidad de composición con *interfaces* especificadas contractualmente y *dependencias contextuales* explícitas.

**Def. II**: Una **interfaz** es un conjunto de operaciones con nombre que pueden ser invocadas por los clientes.

**Def. III**: Las **dependencias contextuales** son especificaciones de lo que el entorno en el que se usa el componente deberá proporcionar, de modo que los componentes puedan funcionar.

Szyperski aporta también características básicas que a su juicio tiene un componente, haciendo énfasis en aspectos comerciales:

**Def. IV**: Las propiedades características de los componentes son:

- Un componente es una unidad de despliegue independiente.
- Un componente es una unidad de composición para terceros.
- Un componente no tiene estado persistente.

Podemos incluir aquí otras definiciones complementarias, debidas a autores también influyentes:

**Def. V**: Un **componente software** es una parte física y reemplazable de un sistema conforme a (y que proporciona la realización de) un conjunto de interfaces [BJRR98].

**Def. VI**: Un **componente software** es un paquete de software que incluye implementación, con una especificación de las interfaces que proporciona y requiere [SW99, p.387].

**Def. VII**: Un **componente software** es una pieza reusable de código y datos en forma binaria que se puede conectar a otros componentes software de otros fabricantes con un esfuerzo relativamente pequeño [Br93].

Como se irá viendo a lo largo de esta tesis, nuestra postura está más cerca de la de D'Souza y Wills (Def. VI). En cualquier caso, estas definiciones se incluyen aquí para futuras referencias (en especial las definiciones de Szyperski, por las razones mencionadas).

## 2.2. Métodos formales

Dentro de diversos movimientos de investigación en Ingeniería del Software aplicada a componentes, se han realizado avances en el uso de técnicas de especificación formal, y en este terreno cabe citar la colaboración entre Jeannette Marie Wing, Edmund M. Clarke

(ambos pertenecientes al CSSRG-CMU que se menciona en 2.7.1), Gary T. Leavens, del Department of Computer Science de la Iowa State University (DCS-ISU), y Amy Moormann Zaremsky, que ha trabajado en Xerox Corporation y en el propio CSSRG-CMU.

Estos autores recogen la motivación última de esta tesis: el hecho de que el ensamblaje de componentes software no puede limitarse a una verificación de congruencia de tipos en las interfaces. En particular, su línea de investigación aborda uno de los problemas clásicos de la reutilización, y es la identificación de un componente ya existente que sea apropiado para una aplicación concreta (se recordará que uno de los obstáculos principales para la reutilización es la imposibilidad de encontrar el componente apropiado aun cuando existe).

Para ello, proponen que la especificación de componentes debe incluir una especificación *formal* de la interfaz de los mismos [LW98], lo que enriquece la información disponible. Se exploran las técnicas de especificación y lo que aportan [CW97, Wi95] (por ejemplo, la resolución del clásico problema de que en las bibliotecas estándar del lenguaje C existen multitud de funciones cuyos argumentos son dos cadenas, y sin embargo tienen misiones y requisitos muy diferentes entre sí), y además de proporcionar una información de mejor calidad, se evalúa cómo estas técnicas pueden abrir las puertas a búsquedas automatizadas y eficientes de los componentes [ZW97, ZW98]. Básicamente, de lo que se trata es de encontrar un buen proceso de *unificación* de una especificación de necesidades con una descripción de un componente disponible.

Estas investigaciones "aplicadas" tienen sus raíces en métodos de especificación formal de propósito general, como la notación Z propuesta por Spivey [Sp89, Sp92] y que ha sido utilizado con frecuencia en otros ámbitos, como en el grupo CSSRG-CMU [Ab96, Ga97].

Las técnicas de especificación formal parecen un campo prometedor de cara a la sistematización del proceso de desarrollo y la verificación de diversas etapas del mismo, y en la literatura se narran muchos casos reales de aplicación con éxito de tales técnicas [CW97], siendo quizás un ejemplo claro el campo de los protocolos de comunicaciones con el uso de lenguajes como SDL, Estelle y Lotos [Tu93, BHS91].

Más adelante (capítulo 3.2) se ofrece una discusión sobre en qué medida los métodos formales en general responden al planteamiento de esta tesis. Durante dicha discusión se ofrece información adicional sobre estos métodos formales (Z, VDM, B, etc.)

## 2.3. Análisis estático e interpretación abstracta

El presente apartado se incluye porque las disciplinas aquí mencionadas resultan de interés fundamental cuando se habla de la verificación o validación de un sistema de software en términos generales.

### Descripción general

Es bien sabido que las (necesarias) actividades de prueba del sistema plantean serias dificultades para su realización. El potencial espacio de estados que un sistema software puede alcanzar es enorme, y a efectos prácticos suele ser imposible recorrerlo por completo. Por tanto, es necesario seleccionar un subconjunto significativo de dicho espacio de estados, basándose en criterios (en gran medida heurísticos) recogidos en la literatura sobre ingeniería del software.

No es reciente la idea de recorrer este espacio de estados de forma analítica. Al fin y al cabo, la resolución analítica de problemas se ha aplicado con éxito en multitud de áreas de la matemática. Sin embargo, la automatización de este análisis en el caso que nos ocupa resulta especialmente difícil; el análisis de programas de ordenador mediante otros programas de ordenador presenta dificultades especiales, e incluso límites absolutos a la computabilidad.

El *análisis de programas* es una técnica que persigue predecir el conjunto de valores o comportamientos que pueden surgir durante la ejecución de un programa, sin tener que realizar su ejecución real en todos los conjuntos de estados posibles. Este problema es, en general, no computable [Ma99] y por ello se utilizan necesariamente aproximaciones.

El análisis de programas tiene sus fuentes originales en el *análisis de flujo de datos* [Ki73], que luego se amplió (inscribiéndolo en un marco más general) en la llamada *interpretación abstracta*.

La **interpretación abstracta** de programas [CC76, Co96, Co96b, JN94, NNH99] es una técnica que permite deducir propiedades de los mismos sin necesidad de recurrir a su ejecución. Se basa en que la semántica de un lenguaje de programación puede ser más o menos precisa, según el nivel de observación elegido. Si se observa el programa sin exigir un alto nivel de detalle, se puede realizar una abstracción de su semántica que, aun siendo menos precisa que la original, en contrapartida es computable. Esencialmente esto consiste en que a las variables no se les asocian valores "concretos", sino "abstractos"; por ejemplo, tratándose de variables enteras se recurre a la asignación de rangos o dominios de valores. El programa se evalúa bajo esa premisa, pero en lugar de realizar su ejecución, las instrucciones del programa se interpretan (aquí entra la *interpretación abstracta*) mediante un mecanismo matemático de cierre transitivo.

Este mecanismo, aplicado a conjuntos infinitos, plantea evidentes problemas; para evitarlos, las propiedades del algoritmo de cierre transitivo se eligen de forma que la interpretación abstracta sea convergente y se estabilice en un número finito de pasos. Gracias a esto, se asegura que el proceso finaliza en tiempo finito y por tanto puede realizarse "en tiempo de compilación". Pero es necesario pagar un precio por ello; los valores abstractos obtenidos son una aproximación, que lleva implícita una posible pérdida de información. Evidentemente, los sistemas de interpretación abstracta se construyen de forma conservadora, para que se obtengan siempre resultados correctos aunque no sean completos.

El **análisis estático** se basa en la interpretación abstracta para, dada la semántica original de un programa, derivar otra semántica aproximada y computable. De este modo, los ordenadores pueden analizar en tiempo de compilación el comportamiento que un programa tendrá en tiempo de ejecución y prever problemas, lo que lógicamente resulta fundamental de cara a la calidad.

La interpretación abstracta, pues, permite realizar deducciones estáticas sobre propiedades dinámicas. La interpretación de las operaciones elementales, así como los valores abstractos que se asocian a las variables, se eligen dependiendo de las propiedades dinámicas que se desee estudiar. Un grupo de considerable importancia en este campo de investigación es el encabezado por Patrick y Radhia Cousot, cuya actividad comenzó ya en los años 1970 [CC76, CC77, CH78]. En estos trabajos se presentan diferentes algoritmos de análisis estático.

Existen multitud de trabajos relacionados con análisis estático de programas e interpretación abstracta; como mero ejemplo, véase en [CF99] una relación de ponencias sobre investigación en este terreno y su aplicación a muy diversos aspectos de la programación, incluyendo cuestiones relacionadas con la sincronización en programas Java, optimizaciones, análisis de grandes programas, etc. Por ejemplo, en [CH78] se ofrecen algoritmos de interpretación abstracta que permiten deducir qué relaciones lineales existen entre las variables de un programa. En [BeC85] (aunque no aparece el término "interpretación abstracta") se realiza también una formalización y se muestran técnicas de análisis basadas en ella, que permiten deducir de forma automática información como:

- dependencias entre variables
- a qué variables afectaría (directa o indirectamente) la modificación de una de ellas
- sentencias que no tienen efecto
- variables no definidas
- estabilidad de variables en bucles (si la variable estable constituye la condición de salida del mismo, se estaría detectando un potencial bucle infinito)

Este enfoque se ha visto convenientemente apoyado en un considerable grado de formalización matemática. En cuanto a su aplicación práctica, tradicionalmente los principales campos de aplicación de la interpretación abstracta son la optimización de código en compiladores [FO75, JN94 p.10] y la verificación de programas para comprobar que no se llega a ningún estado indeseado en los mismos. Diversas ideas de la interpretación abstracta se utilizan, por ejemplo, en la construcción de compiladores con el fin de detectar ciertos errores en tiempo de compilación (uso de variables no inicializadas, por poner un ejemplo). No obstante, el tipo de errores que se detectan no suele ser muy sofisticado; por ejemplo, habitualmente los compiladores no realizan un análisis lo suficientemente profundo como para prever desbordamientos o variables que se salgan de rango.

Existen también trabajos relacionados con la transformación de programas. Otro ejemplo de aplicación práctica es PAG [Ma99, PAG], un generador de analizadores de programas.

Estas técnicas no están necesariamente ligadas al uso de componentes; de hecho, tradicionalmente se han aplicado al código fuente de algún lenguaje de programación, siendo notorio su uso en lenguajes procedimentales clásicos y existiendo también cierta relación con los lenguajes de programación lógica y funcional. Se han iniciado trabajos que relacionan la interpretación abstracta con los componentes software; un ejemplo de esto es el proyecto Daedalus [DAE], titulado *Validation of critical software by static analysis and abstract testing* y que se desarrolla desde 2000 hasta 2002 (dirigido precisamente por Patrick Cousot). Puesto que las técnicas de validación actuales para sistemas concurrentes, de tiempo real y basados en componentes no soportan bien el cambio de escala, este proyecto pretende dar lugar a métodos de uso industrial para las técnicas de análisis estático y prueba abstracta, que serían aplicados en sistemas aeronáuticos (el proyecto está promovido entre otros por el consorcio AIRBUS) y en un futuro a otros sistemas críticos (tales como sistemas médicos o de automoción).

El enfoque que se está dando a este proyecto de interpretación abstracta basada en componentes no surge, sin embargo, del interés por los componentes en sí mismos. El análisis estático *tal cual* nunca es capaz de detectar todos los errores potenciales de un

programa, por razones básicas de indecidibilidad; existe un pequeño porcentaje de posibles errores sobre los cuales el método utilizado no es concluyente. Esto, que en la mayoría de los programas comerciales no es un problema, puede ser inaceptable en software crítico. La precisión del análisis estático puede mejorarse desde el punto de vista teórico, pero en la práctica eso suele implicar un drástico aumento de los recursos computacionales necesarios, que en programas de cierto tamaño puede hacer el análisis inviable. El proyecto Daedalus, entre otras cosas, pretende realizar un análisis *fragmentado*, de modo que se analicen fragmentos de programa de un tamaño que sea abordable mediante las técnicas precisas y costosas; manteniendo pequeño el tamaño de esos fragmentos, estos métodos pueden realizar su análisis, cuyos resultados se combinarán para realizar el análisis global.

## PolySpace

Como ejemplo de un producto comercial para la verificación de programas que se basa en la interpretación abstracta (y que de hecho es fabricado por una empresa creada por algunos de los investigadores que trabajaban con Cousot & Cousot) citamos aquí al verificador de PolySpace Technologies.

El PolySpace Verifier es una herramienta de verificación estática de programas (mediante interpretación abstracta) que se basa en las teorías mencionadas previamente. Se suministran versiones que permiten verificar código fuente en ANSI C y Ada, sin necesidad de modificar en forma alguna dicho código fuente. Este programa realiza su análisis del código fuente de manera supuestamente exhaustiva. A raíz de ese análisis, muestra al usuario (mediante colores) las diversas zonas de código que son seguras, que albergan errores de ejecución potenciales (dependiendo de los datos de entrada u otros factores) y que albergan errores seguros.

Puede verse la pantalla del PolySpace Viewer (el programa que permite examinar los resultados del análisis) en la Ilustración 2. Para el programa example.c, se muestran agrupados (en la parte izquierda) los diversos errores de ejecución; según el color de cada nodo, el usuario puede ver qué tipo de errores se producen en cada categoría. En el ejemplo, los errores de tipo *Pointer Arithmetic* aparecen en rojo, lo que significa que albergan un error seguro; desplegando ese nodo, se ve que hay una alusión en rojo a cierta línea del programa, y pulsando en dicha alusión se puede ver el punto concreto en el que se produce un error (una asignación de un valor a un puntero que apunta a memoria no válida). Pulsando sobre el elemento marcado en el código fuente (en este caso el asterisco), se obtiene una explicación del motivo del error (Ilustración 3).

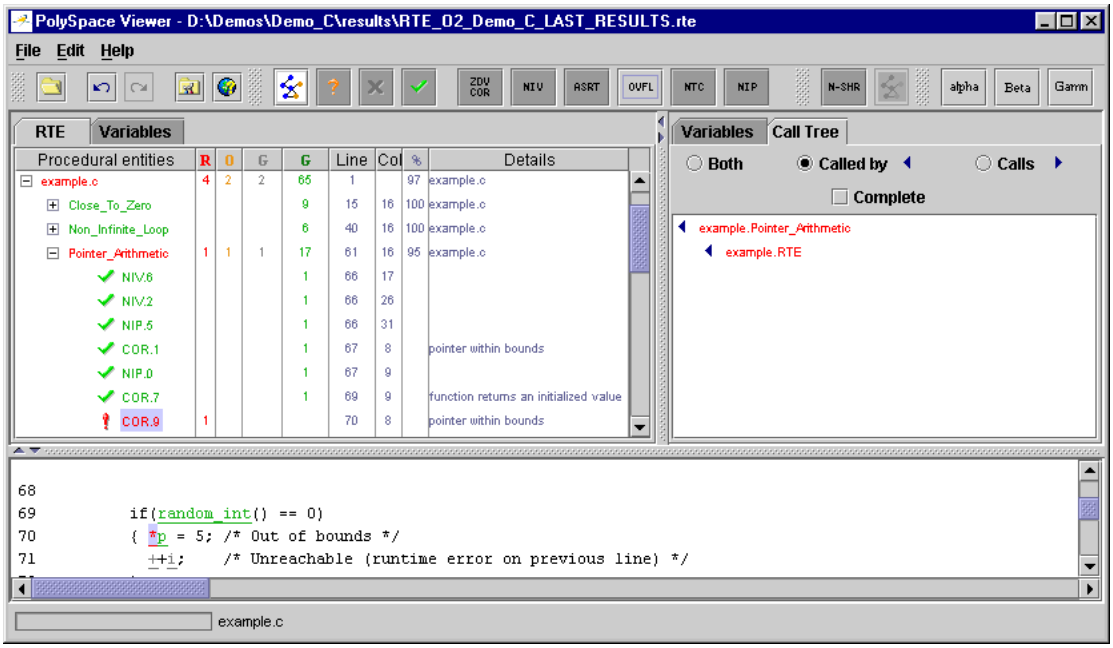



*Ilustración 2. PolySpace Viewer en acción.*

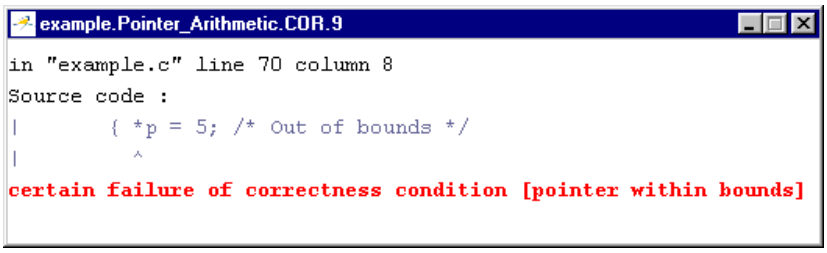



*Ilustración 3. PolySpace Viewer explicando el motivo de un error.*

En aplicación de los conceptos de interpretación abstracta descritos, Verifier construye un modelo abstracto de la aplicación que se analiza, y *propaga* los dominios de las variables de modo que se tiene una vista de todas las ejecuciones posibles. Esta vista se usa para identificar errores de ejecución o de no determinismo.

Esta herramienta afirma dar soporte a aplicaciones de gran tamaño (hasta 150.000 líneas de código), para lo que es necesario realizar un balance entre la precisión con la que se representan los dominios de variables (y los estados del programa) y el tiempo necesario para el análisis.

La ayuda en línea de PolySpace Verifier ofrece algunos comentarios sobre las técnicas ya presentadas, que encaja perfectamente con nuestra descripción:

> *El término Verificación Abstracta es muy amplio, y cubre cualquier herramienta que derive propiedades dinámicas de un programa sin ejecutarlo realmente. La Verificación Estática difiere significativamente de otras técnicas, como la depuración en tiempo de ejecución, en que el análisis que ofrece no se basa en un caso de prueba dado, o en un conjunto de casos de prueba. Las propiedades dinámicas obtenidas en el análisis son ciertas para todas las ejecuciones del sofware.*

> *[...] PolySpace Verifier proporciona un análisis en profundidad, identificando errores de ejecución y posibles conflictos de acceso a datos globales compartidos. En círculos académicos este problema está considerado como muy difícil y frecuentemente se considera no computable en general.*

Los tipos de errores que detecta una herramienta como PolySpace Verifier (en su versión para lenguaje C) son:

- Conflictos de acceso para datos compartidos no protegidos.
- Variables sin inicializar.
- Acceso a través de punteros nulos o fuera de rango.
- Acceso a arrays fuera de rango.
- Operaciones aritméticas no válidas (como divisiones por cero o raíces cuadradas de números negativos).
- Desbordamiento de operaciones aritméticas para enteros y reales.
- Conversiones ilegales de tipos.
- Detección de bucles infinitos, llamadas sin retorno y código inalcanzable, aunque el fabricante advierte de que estos elementos no se detectan de forma exhaustiva.

## 2.4. Técnicas de especificación semántica

En la mayoría de las ocasiones, los aspectos sintácticos de los lenguajes de programación se describen mediante la notación BNF o variantes de la misma, pero la semántica suele especificarse en lenguaje natural [ANSI78, JSGB00]. Esto conlleva que dichas especificaciones pueden resultar ambiguas o poco rigurosas, lo que trae como consecuencia que no sea fácil demostrar propiedades básicas de los programas, validar los compiladores o traductores de forma inequívoca, o incluso generar compiladores de forma automática a partir de la especificación. Para paliar estos problemas de ambigüedad se han desarrollado diversas técnicas de especificación semántica de los lenguajes de programación.

Ninguna de estas técnicas ha llegado a utilizarse en la práctica en un grado notable. Existen diferentes formas de abordar el problema (las describiremos primero muy someramente, para luego justificar la mención a las mismas en esta disertación). La semántica operacional [Pl81] especifica las transiciones elementales de un programa mediante reglas de inferencia definidas por inducción sobre su estructura. En la semántica denotacional, el comportamiento del programa se describe modelando los significados a través de entidades matemáticas básicas. C. Strachey [St67] propuso inicialmente estudiar los lenguajes de programación desde una perspectiva denotacional, mediante $\lambda$–cálculo. Posteriormente se pasó a utilizar teoría de dominios a raíz de la colaboración en 1969 entre Stachey y D. Scott. La semántica axiomática [Hoa69] consiste en definir reglas de inferencia que caracterizan las propiedades de las diferentes construcciones del lenguaje, y tiene una notable relación con el uso de precondiciones y postcondiciones (véanse en esta disertación los apartados sobre programación por contratos).

La semántica algebraica [GM96] surge a finales de la década de 1970, a partir de trabajos desarrollados para la especificación algebraica de tipos de datos. Permite definir una estructura matemática de forma abstracta, junto con las propiedades que debe cumplir. Existen diversos lenguajes de especificación algebraica, como ASL, Larch, ACT ONE, Clear, etc. En semántica algebraica se definen primero los géneros (*sorts*) que se están especificando, las operaciones sobre los mismos y sus funcionalidades; a partir de ahí se definen axiomas, sentencias lógicas que describen el comportamiento de las operaciones (en forma de ecuaciones).

Ya se ha dicho que el fin principal que persiguen estas técnicas es especificar la semántica de lenguajes de programación. Puede parecer que es un problema sin relación con el que nos ocupa, pero lo cierto es que los componentes software están implementados en algún lenguaje de programación, cosa que claramente se aprecia en el comportamiento externo del mismo. De hecho, en gran medida los componentes se identifican con la noción de objeto (operaciones incluidas), y otras técnicas de especificación de interfaces, como la programación por contratos, se apoyan en lenguajes de programación para describir el comportamiento de un componente.

La combinación de componentes entre sí también guarda un claro parecido con utilizar un lenguaje de programación, y existen lenguajes de composición de componentes o lenguajes de descripción de arquitectura (ADLs) que se mencionan con apartado propio en esta revisión; la especificación de la semántica de un lenguaje de programación, sus operadores y construcciones básicas, guarda gran similitud con la especificación de la semántica de las operaciones de un componente. Por tanto, parece razonable pensar que las técnicas de especificación para lenguajes admitirían una adaptación para su uso con componentes software.

Existen, además, variantes de las técnicas de especificación de semántica que pretenden ser modulares y legibles, mostrando estos dos aspectos una clara coincidencia con algunos de los objetivos de esta tesis. La semántica de acción, propuesta por Mosses [Ms92] pretende precisamente crear especificaciones más legibles, modulares y utilizables. La semántica de un lenguaje se especifica mediante *acciones*, que expresan computaciones. Para ello se emplea una notación que simula un lenguaje natural (por supuesto, restringido). Internamente, esta notación se transforma en semántica operacional estructurada. Se parte de acciones primitivas y se utilizan ciertos *combinadores de acciones* para construir acciones más complejas.

La semántica monádica modular [LH96, LHJ95] surge a partir de la utilización de mónadas y transformadores de mónadas para describir una semántica denotacional modular. Una mónada separa una computación del valor devuelto por dicha computación. Con dicha separación se favorece la modularidad semántica, pudiendo especificarse características computacionales de los lenguajes de forma separada. Investigaciones en las que ha participado el autor de esta tesis se han centrado en la semántica monádica reutilizable; esta ofrece la posibilidad de "construir" la semántica de un lenguaje de programación mediante la combinación de módulos de especificación que de este modo son reutilizables. Se ha desarrollado un sistema de prototipado de lenguajes que ha sido aplicado a lenguajes imperativos [LCLG01], funcionales [LLCC01], orientados a objetos [Lb01] y de programación lógica [LCLC01]. Estas especificaciones son modulares y reutilizables, en el sentido de que, dada una especificación de un lenguaje de programación lógica, por ejemplo, es posible dotarlo de la semántica para operaciones aritméticas añadiendo a la

especificación un módulo que describe dicha semántica. El intérprete capaz de procesar el lenguaje descrito puede generarse de forma automática a partir de dicha descripción.

Dado este panorama, parece que la especificación semántica derivada de la semántica de acción y posteriormente de la semántica monádica reutilizable puede ser una herramienta útil para describir diversos componentes, de forma modular y verificable.

# 2.5. Especificación de procesos

Bajo este apartado se describen algunas técnicas desarrolladas en relación con la especificación de procesos como entidades que se comunican a través de canales, y con la idea de *álgebra de procesos*. Este estudio no es, en modo alguno, exhaustivo; se trata sólo de recoger algunas de las tendencias en este campo que estando más relacionadas con los componentes software sean a la vez representativas del propósito o características de otras que no aparecen mencionadas aquí.

## 2.5.1. CSP y otras álgebras de procesos

CSP (*Communicating Sequential Processes*) [Hoa85, Rs97, CSPOx] fue descrito por primera vez en un artículo de C.A.R. Hoare, en 1978. Las ideas básicas de este artículo se fueron refinando posteriormente, hasta conseguir una versión más flexible de la notación, que constituyó la base del libro fundamental sobre CSP [Hoa85]. Sucesivamente, han ido apareciendo versiones adaptadas de CSP. TCSP, de Steve Schneider [Sc99], añade el tratamiento del tiempo a CSP. CSPP, de A. Lawrence, introduce el concepto de prioridad para describir sistemas con recursos limitados; HCSP, del mismo autor, extiende CSPP con el fin de describir sistemas de hardware que involucran estados, relojes, etc.

CSP es un lenguaje formal. Como sugiere su nombre, CSP permite describir los sistemas como un conjunto de procesos (que harían el papel de componentes básicos) que operan independientemente y se comunican entre sí a través de canales bien definidos. La restricción de que los procesos fuesen secuenciales se eliminó en el proceso de refinamiento que comenzó en 1978 y culminó en 1985, pero el nombre ya estaba asentado y se conservó. CSP es una notación apropiada para diversos tipos de problemas, sobre todo los que involucran paralelismo de manera inherente.

Los procesos se describen en términos de *entidades* que interactúan mediante *eventos*. Es posible convertir las interacciones entre ciertos procesos en privadas y dejar como públicas las interacciones con el exterior; de este modo se pueden componer procesos y se obtiene un nuevo proceso como resultado.

Para representar un proceso, este se describe en términos de los eventos en los que puede participar, y los grupos de procesos se pueden describir según las *trazas* de los eventos en los que participaron. Por ejemplo, una máquina de venta automática (MV) que acepta una moneda, suministra una lata y vuelve al estado inicial se puede describir mediante:

$$MV = moneda \rightarrow lata \rightarrow MV$$

Las trazas de MV serían:

<moneda, lata>

<moneda, lata, moneda, lata>

< moneda, lata, moneda, lata, moneda, lata>

...

Sobre estas ideas básicas se desarrolla una notación formal para describir los procesos y su interacción. Basándose en las ideas de CSP, ingenieros de la compañía INMOS (hoy desaparecida; sus productos pasaron a ser gestionados por STMicroelectronics, [STM]) desarrollaron, junto con el hardware de su producto el Transputer, el lenguaje occam [IN88, JG88], para la programación de dicho dispositivo de procesamiento concurrente. La última versión de occam en ver la luz fue la 2.1, y aunque se definió una versión 3 no ha sido implementada y es improbable que lo sea, puesto que actualmente se han encontrado mejores alternativas para resolver los problemas que el lenguaje abordaba. Recientemente, algunos de los principios de occam se han transportado a bibliotecas de Java (JCSP y CCJ).

El modelo CSP ha tenido influencia posteriormente en muchos otros desarrollos, frecuentemente dentro del campo de la concurrencia y paralelismo. Otras álgebras de procesos que cabe citar son CCS (*Calculus of Communicating Systems*) de Robin Milner [Ml89] (para una comparación entre CCS y CSP véase [Gl86]), y ACP (*Algebra of Communicating Processes*) de Bergstra y Klop, que fueron quienes acuñaron el término *álgebra de procesos* en 1982.

### 2.5.2. π-cálculo

El π-cálculo de Robin Milner [MPW89, Ml93, Ml99, SWa01] es un cálculo de sistemas que se comunican entre sí; originalmente fue descrito como un *cálculo de procesos móviles*. Una de sus características principales es que permite modelar y expresar procesos cuya estructura cambia. Los procesos representados en π-cálculo no sólo pueden estar interconectados de manera arbitraria, sino que los actos de comunicación que realizan pueden transmitir información sobre enlaces, de modo que la estructura de interconexión cambie gracias a esa comunicación (de ahí las alusiones a la movilidad).

π-cálculo nació como una extensión del álgebra de procesos CCS [Ml89], aprovechando trabajos de Uffe Engberg y Mogens Nielsen [EN86] que añadieron la denominada *movilidad* a CCS conservando sus propiedades algebraicas. π-cálculo simplifica la propuesta de Engberg y Nielsen conservando sus capacidades.

Haciendo una breve descripción, el π-cálculo consta de los siguientes elementos. Se supone un conjunto infinito de nombres $\mathcal{N}$, donde $v, w, x, y, z$ representan nombres. Estos nombres pueden aludir tanto al sujeto de la comunicación (el canal) como al objeto de la misma (el parámetro que se transmite por el canal); esta flexibilidad y unificación en el uso de nombres es una de las características básicas que permiten incorporar el concepto de movilidad. También se tiene un conjunto $\mathcal{K}$ de *identificadores de agente*, cada uno de los cuales tiene una aridad (un entero ≥0). *A, B, C, ...* representan identificadores de agente. *P, Q, R, ...* representan las *expresiones de proceso o agente*, que son de seis tipos:

1. Suma $\Sigma P_i$, una suma finita, en la que el agente se comporta como uno de los $P_i$. La suma vacía se representa 0 y se denomina *inacción*; es un agente que no hace nada.
2. Una forma prefija $\overline{y}x.P$, $y(x).P$ ó $\tau.P$. La primera se denomina un *prefijo negativo*. Puede verse $\overline{y}$ como un puerto de salida del agente; emite el nombre $x$ por el puerto $\overline{y}$ y a continuación se comporta como $P$. La segunda es un prefijo positivo; $y$ sería

un puerto de entrada al agente; $y(x).P$ recibe un nombre arbitrario $z$ en el puerto $y$, y después se comporta como $P\{z/x\}$. $\tau$ se denomina un *prefijo silencioso*. $\tau.P$ realiza la *acción silenciosa* y después se comporta como $P$.

3. Una composición $P_1 \mid P_2$. Este agente consta de $P_1$ y $P_2$ actuando en paralelo. Los agentes pueden actuar independientemente. También, una acción de salida de $P_1$ puede sincronizarse con una acción de entrada de $P_2$, dando lugar a la acción silenciosa $\tau$ en el agente compuesto $P_1 \mid P_2$.
4. Una restricción $(y)P$. Este agente se comporta como P, pero las acciones en puertos $\overline{y}$ o $y$ están prohibidas. No obstante, es posible la comunicación en el enlace $y$ entre componentes de $P$; esta restricción vendría a significar que el puerto $y$ es privado a los componentes de $P$.
5. Un emparejamiento $[x=y]P$. Este agente se comporta como $P$ si los nombres $x$ e $y$ son iguales; si no lo son, se comporta como 0.
6. Un agente definido $A(y_1, \ldots, y_n)$. Para cualquier identificador de agente A, debe haber una única ecuación que lo define, $A(y_1, \ldots, y_n)=P$. Estas ecuaciones de definición introducen la recursividad, puesto que P puede contener cualquier identificador de agente, incluyendo el propio A.

Basándose en estos conceptos básicos, Milner et al definen unas reglas de transición y equivalencia. Por ejemplo:

$$\overline{y}x.P \mid y(z).Q \rightarrow P \mid Q\{x/z\}$$

Es decir, el primer agente $\overline{y}x.P$ emite el nombre $x$ por su puerto $y$, y simultáneamente se tiene un agente $y(z).Q$ que recibe un nombre por su puerto $y$. Esto da como resultado que el primer agente, tras emitir el nombre $x$, se comporte como $P$, y el segundo agente, tras recibir ese nombre en su puerto $y$, pase a comportarse como $Q$ (habiéndose sustituido todas las apariciones del nombre libre $z$ por su valor actual $x$).

Como ya se ha señalado, uno de los puntos fuertes del π-cálculo es que los propios nombres de puerto pueden viajar como cualquier otro valor; es decir, a la hora de servir de parámetros de una comunicación, no se establece una diferencia explícita entre los nombres y los puertos. Esto hace que sea posible denotar el acto de comunicación por el que un proceso le suministra un enlace a otro proceso, y por tanto la estructura de comunicaciones cambia a raíz de ese acto de comunicación. Otros modelos de movilidad citados en [MPW89] consiguen el cambio en la estructura del sistema permitiendo el paso de los propios procesos como parámetros *por valor*; pero esto conlleva una duplicación de procesos que Milner et al deseaban evitar como efecto lateral de la comunicación. Por eso, esta homogeneización del concepto de *nombre* que engloba también a los propios puertos permite el paso de una suerte de *referencias* a los procesos (sus puertos).

El π-cálculo representa una idea simple pero potente. Su principal campo de aplicación parece ser el de los procesos concurrentes, su modelado y verificación. La idea básica detrás del π-cálculo es que este represente para los procesos lo que el λ-cálculo representó para las funciones, y que los lenguajes de alto nivel para programación concurrente se apoyen en el π-cálculo como los lenguajes funcionales se apoyaron en el λ-cálculo; véase la propuesta de

[Pi95]. De hecho, la propuesta del π-cálculo es representar incluso las propias estructuras de datos y los tipos igual que los procesos [SWa01].

El π-cálculo se menciona y describe aquí porque es una forma de representación relativamente conocida (aunque su uso no esté extendido a nivel de producción sí representa un indudable referente en el campo de la investigación, y está considerado como una forma mejorada de abordar los problemas de los tradicionales modelos CSP y CCS) y su modelo introduce entidades con entradas y salidas que se comunican, de modo que existe una semejanza con nuestros objetivos respecto a los componentes. Por otra parte, se trata de una notación formal con el potencial de realizar ciertos tipos de verificaciones automatizadas. Ambas vertientes han sido consideradas en ocasiones para su aplicación al ámbito de los componentes software tradicionales, con fines parcialmente similares a los perseguidos aquí [CFPTV01].

### 2.5.3. π*L*-cálculo

En este apartado se presenta un derivado del π-cálculo, denominado π*L*-cálculo [ALSN00, LAN00]. El π-cálculo basa su comunicación en el uso de tuplas. Esto restringe la extensibilidad y la reutilización, debido a la existencia de parámetros ligados a una posición [ALSN00, p. 5; LAN00, p. 2]. Los procesos emisor y receptor deben estar de acuerdo en el número de nombres comunicados, así como en su interpretación. Este esquema es demasiado rígido, ya que las extensiones introducidas en una parte deben transmitirse a otras. Este efecto puede reducirse, si se reemplaza la comunicación poliádica de tuplas por la comunicación monádica de *forms* (que aquí denominaremos *formularios*).

El π*L*-cálculo, por su parte, es inherentemente extensible; nace como un derivado de un fragmento asíncrono del π-cálculo. Básicamente, se reemplaza la comunicación de nombres (en el π-cálculo monádico) o tuplas de nombres (en el π-cálculo poliádico) por la comunicación de los llamados *formularios*, que son un tipo especial de registros extensibles. Las ideas fundamentales del π-cálculo (procesos, canales, puertos) se mantienen aquí, pero mientras que en π-cálculo básico los nombres se utilizan indistintamente como sujetos y como objetos de la comunicación (es decir, como nombres de puerto y como parámetros) en el π*L*-cálculo los nombres son sólo sujetos (nombres de puerto), y los objetos de la comunicación son siempre formularios. El formulario es una correspondencia (*mapping*) finita entre un conjunto infinito de etiquetas y el conjunto infinito de nombres aumentado con el valor vacío. Puede preverse que el uso de estos valores más ricos añade posibilidades y flexibilidad respecto al π-cálculo original; además, al identificarse los valores mediante nombres en el formulario y no mediante una correspondencia directa en posición y número, los problemas de extensibilidad mencionados para el π-cálculo decrecen.

En general, es posible trasladar una descripción de un sistema en π-cálculo a π*L*-cálculo y viceversa, conservando en ambos sentidos un comportamiento matemáticamente equivalente.

El π*L*-cálculo se diseñó para servir como herramienta a la hora de razonar sobre concurrencia y comunicación, pero resulta de un nivel extremadamente bajo como lenguaje de programación en la práctica [ALSN00, p. 7].

## 2.5.4. Lenguajes derivados

Se mencionan aquí diversas líneas de investigación en lenguajes y notaciones que se derivan del trabajo en la especificación de procesos concurrentes.

### Lenguajes de composición

Los denominados *lenguajes de composición* [NM95] permiten construir aplicaciones simplemente conectando componentes. La evolución de una aplicación de este tipo se consigue cambiando los componentes involucrados y / o las conexiones entre ellos. Para poder hacerlo, el lenguaje de composición reúne las siguientes ideas:

- Un **estilo arquitectónico** (véase 2.7) formaliza las interfaces, los conectores y las reglas de composición de los componentes.
- Un **guión** (*script*) define y especifica una composición concreta.
- Las conexiones se implementan mediante **abstracciones de coordinación**.
- Las **abstracciones de unión** (*glue abstractions*) adaptan y proporcionan envoltorios para los componentes que no se ajustan al estilo arquitectónico utilizado.

El lenguaje de composición suele ser un lenguaje de alto nivel, y de lo anterior se deduce que su propósito práctico es combinar componentes software, construidos en otros lenguajes de desarrollo convencionales. Un lenguaje de composición no es simplemente un lenguaje de *scripting*; va más allá que estos, puesto que el lenguaje de composición no debería estar orientado hacia un paradigma específico de programación de guiones, sino que debe permitir combinar componentes según diferentes estilos de composición. Las técnicas de orientación a objetos, por ejemplo, permiten tratar las aplicaciones como composiciones de objetos que colaboran, pero ofrecen un soporte limitado para otras abstracciones de granularidad más gruesa o fina que los propios objetos. Un lenguaje de composición no pone el énfasis en la programación y herencia de clases, sino en la especificación y composición de componentes. Los objetos se ven como procesos, y los componentes son abstracciones sobre el espacio de objetos; la aplicación es una composición explícita de componentes software.

El punto de vista de estos lenguajes de composición se resume en los siguientes requisitos [NM95]:

- **Topología abierta**: Las aplicaciones abiertas son inherentemente concurrentes y distribuidas.
- **Heterogeneidad**: Las aplicaciones deben ejecutarse en diversas plataformas hardware y software.
- **Requisitos cambiantes**: Los requisitos de una aplicación no están fijados desde el principio, por lo que se necesita una arquitectura flexible para acomodar los requisitos que evolucionan.

Estos requisitos dan pie a las siguientes características técnicas:

- Encapsulamiento.
- Objetos como procesos: Los objetos pueden ser activos o pasivos, locales o remotos, simples o compuestos, pero todos los objetos pueden verse como procesos.
- Componentes como abstracciones: Los componentes son abstracciones software (de orden potencialmente superior) que se componen de diversas formas para producir aplicaciones.
- Compatibilidad para la conexión: Se necesita un modelo de tipos que involucre a objetos y componentes para razonar sobre enlaces y composiciones válidos.
- Modelo de objetos formal: Se necesita un modelo de objetos estándar.
- Escalabilidad: El uso del lenguaje debería ser aplicable a sistemas grandes y pequeños, y tanto en un uso muy dinámico con información de tipos incompleta como en un uso compilado y altamente optimizado.

**Piccola**

Piccola [Piccola, ALSN00, AN01] es un lenguaje de composición experimental. Se define mediante una fina capa sintáctica sobre un núcleo semántico que se basa en el $\pi\mathcal{L}$-cálculo (véase 2.5.3), que a su vez se basa en el $\pi$-cálculo de Robin Milner et al (véase 2.5.2). El nombre Piccola es una contracción de *PI-Calculus based COmposition LAnguage*, lenguaje de composición basado en $\pi$-cálculo.

Este lenguaje ha sido desarrollado por el Software Composition Group de la Universidad de Berna (Suiza) y, como lenguaje de composición que es, ha sido diseñado para facilitar la definición de conectores de alto nivel para componer y coordinar componentes software escritos en diferentes lenguajes. Han aparecido ya diversas versiones de Piccola (Piccola1, Piccola2, y Piccola3 que está aún en desarrollo). JPiccola es una implementación de Piccola en Java.

Como ya se ha dicho, el $\pi\mathcal{L}$-cálculo resulta de un nivel de abstracción demasiado bajo para utilizarlo como lenguaje de programación operativo. Además, el estilo natural de interacción definido por $\pi\mathcal{L}$-cálculo es el de comunicación unidireccional a través de canales; se pueden representar otras formas de comunicación (como la basada en eventos) pero su codificación resulta frecuentemente poco elegante. Esta es la motivación de un lenguaje como Piccola: ofrecer construcciones que simplifiquen estas representaciones. Tales construcciones incluyen funciones, operadores infijos que den soporte a una noción algebraica de estilo arquitectónico, y la noción explícita de un contexto (dinámico) para encapsular los servicios requeridos.

A partir de estas construcciones, se pueden definir abstracciones de alto nivel como funciones de biblioteca, al igual que CLOS se definió sobre Common Lisp. Las abstracciones de Piccola, en vez de sobre funciones y listas como en Lisp, se construyen sobre los conceptos formales de agentes, formularios y canales.

Aunque no es objeto de este documento ofrecer información detallada de Piccola, sólo a efectos orientativos se puede describir someramente el lenguaje. La sintaxis general de

Piccola es parecida a la de Python o Haskell en el sentido de que para delimitar formularios o bloques se utilizan las marcas de fin de línea y la indentación, en lugar de etiquetas o llaves. Los elementos básicos del lenguaje se pueden consultar en la Ilustración 4. Debe tenerse en cuenta que muchas operaciones básicas del $\pi\mathcal{L}$-cálculo, como por ejemplo la creación de un nuevo canal o el prefijo de entrada o salida, se representan mediante abstracciones funcionales predefinidas.

| Expresión | Significado |
|---|---|
| `ident = e` | Asigna la expresión de formulario e al nombre `ident`. |
| `export ident = e` | Extiende el contexto actual con la asignación `ident = e`. |
| `e.ident` | Devuelve el valor al que está asignada la etiqueta `ident` en el formulario e. |
| `Def ident(ident`$_1$`)...(ident`$_n$`) = e` | Define una abstracción parametrizada sobre la expresión de formulario `e` (dicho de otro modo, define una función `ident` con los parámetros formales `ident`$_1$`, ..., ident`$_n$). |
| `return e` | Devuelve la expresión `e`. |
| `Run e` | Invoca una función denotada por la expresión de formulario `e` asíncronamente (es decir, no devuelve un resultado). |
| `ident (e`$_1$`)...(e`$_n$`) [in e`$_m$`]` | Invoca la función `ident` con los parámetros reales `e`$_1$`, ..., e`$_n$ síncronamente. Si se incluye `in e`$_m$, se utilizará `e`$_m$ como contexto, y si no se utilizará el contexto actual. |
| Operadores infijos | Los operadores como +, -, \|, > son recursos sintácticos para denotar ciertas funciones. |

*Ilustración 4. Los elementos básicos del lenguaje Piccola.*

**Pict**

El desarrollo del lenguaje Pict [PT97] comenzó en la Universidad de Edimburgo en 1992. Pict no es un lenguaje de composición, pero se menciona aquí por su relación directa con el π-cálculo de Robin Milner. De hecho, una primera motivación para el proyecto fue que no había precedentes de tomar la comunicación como único mecanismo de computación; los lenguajes que se habían desarrollado con base en el π-cálculo solían combinar dicha teoría con un núcleo de lenguaje funcional. Benjamin C. Pierce et al deseaban diseñar e implementar un lenguaje concurrente de alto nivel apoyándose exclusivamente en las primitivas del π-cálculo.

El hecho de compilar un lenguaje basándose exclusivamente en procesos que se comunican planteaba importantes problemas de generación de código. Para conseguir una eficiencia aceptable en aplicaciones reales, era necesario implementar la creación de procesos, cambio de contexto y comunicación por canales de manera extremadamente eficiente, ya que estas operaciones eran el mecanismo fundamental de computación en π-cálculo.

Otro objetivo de Pict era explorar la aplicabilidad práctica de trabajos teóricos previos de los autores en el campo de los sistemas de tipos para π-cálculo. Esto obligaba también a abordar problemas de verificación de tipos.

Como resumen, las motivaciones del proyecto Pict eran responder a las siguientes preguntas:

1. ¿Cómo se puede programar en π-cálculo? ¿Qué tipo de lenguaje de alto nivel se puede construir sobre el π-cálculo?
2. ¿Qué tipos de objetos concurrentes surgen al desarrollarlo?
3. ¿Se puede implementar el π-cálculo de manera eficiente?
4. ¿Se puede diseñar un sistema de tipos práctico para el π-cálculo combinando subtipos y polimorfismo de orden superior?

Como conclusión del desarrollo de Pict, cabe citar que el π-cálculo puede verse como un tipo de "código máquina concurrente": simple, flexible, y que se puede implementar eficientemente, ofreciendo un código objeto apropiado para la compilación de otros lenguajes de nivel más alto.

Existen, aparte de Pict, otros muchos tipos de lenguajes concurrentes de alto nivel que no mencionaremos aquí por salirse del ámbito que nos ocupa y por estar suficientemente representados por Pict.

# 2.6. Contratos

¿Por qué merece la pena detenerse en el campo de los contratos en esta disertación? Hay una razón de peso. Uno de los factores que motivaron el planteamiento de esta tesis fue el llegar a la conclusión de que las interfaces de los componentes (tal como se entiende el término "interfaces" habitualmente, es decir, la simple enumeración de funciones con sus signaturas) eran un mecanismo claramente insuficiente para caracterizar cuándo la conexión entre componentes es correcta, cuándo los componentes *encajan*. Lo dicen D'Souza y Wills en [SW99, p.xix]:

> *Las llamadas a función que se describen sólo por las signaturas de sus parámetros no dicen suficiente sobre el comportamiento esperado. Los lenguajes de programación no proporcionan esta posibilidad porque no se espera que representen diseños; pero necesitamos escribir descripciones de interfaz precisas.*

En el mundo del software, los contratos pueden definirse de manera informal como *una interfaz más su especificación* [Sz97, pág. 92]. En lo básico, esta definición es indudablemente un paso en la dirección que se propone en esta tesis; la necesidad de completar las fronteras de los componentes con información sobre sus requisitos y / o su funcionamiento, mucho más allá de la simple descripción de signaturas.

## 2.6.1. Programación por contratos: Meyer

Al hablar de componentes y de verificación, no se puede obviar el influyente trabajo de Bertrand Meyer como promotor, quizás el más popular, de la llamada "programación por contratos" [Me99]. Meyer, además de haber sentado cátedra definiendo prácticas

recomendables en el desarrollo orientado a objetos (y de haber trabajado previamente en métodos formales, participando en las primeras versiones de la notación Z), propuso la utilización de precondiciones, postcondiciones e invariantes en las clases, de modo que la corrección del estado de los diferentes objetos se verificase constantemente durante la ejecución de un programa. Aunque como ya se ha dicho no hay por qué identificar "componente" con "objeto" (y en esta tesis especialmente, ya que "componente" es una noción tremendamente genérica) estas ideas tienen relevancia como precedentes (en ciertos aspectos).

De forma introductoria, diremos que Meyer apunta que un elemento de software no es correcto ni incorrecto de por sí; *es correcto si se comporta de acuerdo a su especificación*. Partiendo de esa base, lo primero que se necesita para verificar el software es *una especificación*. Como paso siguiente, Meyer propone incluir esta especificación como parte intrínseca de la implementación. Tal especificación estará formada por expresiones booleanas, que indiquen si ciertos elementos de software cumplen ciertas propiedades; estas expresiones pueden ser precondiciones, postcondiciones e invariantes.

Las precondiciones son un conjunto de condiciones que un método de una clase exige como previas para ejecutarse de forma correcta; las postcondiciones están constituidas por aquello que el método garantiza que ocurrirá tras su ejecución (una *promesa* sobre el estado en que queda el objeto) siempre y cuando se hayan cumplido las precondiciones que exigía, y los invariantes son condiciones sobre el estado del objeto que deben cumplirse en todo momento (tras la ejecución de cualquier método de la clase, tenga o no pre/postcondiciones) para que dicho objeto se considere en un estado correcto. Estas diversas verificaciones son herederas de un recurso mucho más primitivo y presente en el lenguaje C, las cláusulas *assert*. Meyer plasmó estas ideas y otras en el lenguaje de programación Eiffel, del que es autor.

El concepto de contrato surge del hecho de que cada método exige unas precondiciones, y "a cambio" garantiza que se cumplirán unas postcondiciones. De este modo, el estado del objeto está controlado en todo momento, y en la práctica se garantiza que se mantiene coherente. Si se violan los términos de alguno de estos contratos, o de algún invariante, se producirá un error en tiempo de ejecución. Si se incumplen las precondiciones es "culpa" del cliente del objeto (que incumple los prerrequisitos), y si se cumplieron las precondiciones pero se incumplen las postcondiciones es "culpa" del propio objeto (que incumple sus promesas).

La comprobación de pre y postcondiciones es una construcción del lenguaje; básicamente, se realiza mediante expresiones booleanas que no alteran el estado del objeto. Como extensión necesaria en la práctica, Meyer introduce la posibilidad de utilizar *funciones*; estas contendrían instrucciones convencionales (consulta de atributos de los objetos, llamadas a otras funciones, etc.) que tampoco alterarían el estado del objeto, pero ofrecerían una potencia expresiva mucho mayor. Por ejemplo, si se debe comprobar que el número más grande de un array está en la primera posición de este, el expresarlo de forma directa con las operaciones básicas resulta casi imposible, mientras que con una función se puede construir el bucle correspondiente.

Estos contratos en forma de pre/postcondiciones cumplen un papel múltiple:

- En primer lugar, son un método de especificación, de expresar cómo debe comportarse un elemento de software. Aun cuando no tengan ningún efecto en la

práctica y se consideren meros comentarios, el expresar la especificación en una notación formal –como lo es un lenguaje de programación- ayuda a construir software correcto [Me99, pág. 315].

- En segundo lugar, estas construcciones se verifican constantemente en tiempo de ejecución (constituyen código ejecutable). Por tanto, sirven para vigilar en todo momento que el sistema funciona con arreglo a su especificación.

Puesto que la verificación durante la ejecución puede resultar costosa, en los compiladores de Eiffel es posible desactivar tal verificación cuando va a generarse el código que se utilizará en producción.

Las ideas de Meyer sobre la programación por contratos (y muchas de sus recomendaciones sobre construcción de software orientado a objetos) han tenido una gran influencia en la comunidad del desarrollo de software. Como se ha expuesto al principio de este apartado, estas ideas tienen una clara relación con el caso que nos ocupa, en el sentido de que considerando "componentes" a los objetos o clases (dependiendo del nivel de abstracción), estos contratos constituyen al fin y al cabo una forma de verificar las conexiones entre dichos componentes, de una forma similar en algunos aspectos a la descrita aquí.

### 2.6.2. Contratos bilaterales: Wirfs-Brock, Reenskaug et al

Es conocida la aportación de Rebecca Wirfs-Brock en el campo de las técnicas de análisis orientado a objetos, concretamente recogiendo el concepto de las fichas CRC (Clases, Responsabilidades y Colaboradores) introducido por Beck y Cunningham [BC89]. Bajo este prisma, los trabajos de Wirfs-Brock, Wilkerson y Wiener plantean otra noción de "contrato" [WW89, WWW90]. En su terminología cliente/servidor, consideran que los *servidores* son entidades que suministran servicios, y los servicios individuales son las ya mencionadas *responsabilidades*. De este modo, cada clase se considera una "caja negra" que ofrece servicios.

El interés se centra en los objetos individuales, y en las responsabilidades que la clase define. De este modo, la **colaboración** se define como **una petición de un cliente a un servidor** para que el cliente pueda cumplir alguna de sus propias responsabilidades. Esto puede implicar que se necesiten varias colaboraciones para dar soporte a una sola responsabilidad, es decir, un cliente puede realizar varias peticiones a un servidor en el cumplimiento de dicha responsabilidad. Los autores recogen este conjunto de colaboraciones (sólo si es significativo) en un "contrato".

Por tanto, para Wirfs Brock et al un contrato es un conjunto de una o más peticiones de un cliente a un servidor, en esa sola dirección. Esta es una clara diferencia respecto al contrato según Meyer, que resulta ser un conjunto de derechos y deberes *unilaterales*, por parte de una sola clase. En la misma línea que Wirfs-Brock se inscriben los trabajos de Trygve Reenskaug [RWA96] que también identifican que al modelar un sistema orientado a objetos es esencial no dejar que los datos dirijan el proceso, sino que lo hagan las responsabilidades o roles que el sistema debe ver satisfechos.

### 2.6.3. Contratos de reutilización (Vrije Universiteit Brussel)

Aunque lo más habitual es que el término "contrato" en programación se asocie a las ideas de Bertrand Meyer (o Wirfs-Brock et al) en virtud de la mayor difusión de las mismas,

existen otras interpretaciones que tienen más interés en esta tesis. Entre ellas se encuentra la desarrollada en el Programming Technology Lab del Computer Science Department en la Vrije Universiteit Brussel (en adelante, PTL-VUB). Este grupo de investigación (Patrick Steyaert, Theo D'Hondt, Koen D'Hondt, Kim Mens, Carine Lucas entre otros) ha trabajado en lo que se denominan *contratos de reutilización*.

El interés de estos investigadores se centra en los problemas que surgen en la evolución de los componentes. Aun suponiendo que se tiene un sistema construido mediante el ensamblaje de componentes, cuando estos componentes evolucionan, mejoran, o se modifican de cualquier forma, hay un alto grado de incertidumbre sobre el efecto que tales modificaciones tendrá en el conjunto del sistema, y si un componente puede o no sustituirse por una nueva versión. Ocurre con frecuencia que el nuevo componente no se comporta exactamente como los demás esperaban, y se rompen suposiciones básicas que unos componentes realizaban respecto al comportamiento de otros. El resultado es que un sistema que estaba en funcionamiento deja de funcionar.

Con las descripciones habituales de interfaces (que ya se han criticado en esta disertación) este problema es difícil de atajar, puesto que tales descripciones no dicen prácticamente nada sobre cómo interaccionan unos componentes con los otros. En vista de esto, el PTL-VUB propone el uso de los llamados *contratos de reutilización* para avanzar un paso hacia la resolución de este problema.

Habitualmente, la información sobre interoperabilidad se publica en documentación adicional, porque los lenguajes de descripción de interfaces más extendidos no permiten hacerlo. Murer, Scherer y Würtz [MSW97] establecen que debe haber tres niveles de información sobre interoperabilidad:

- Nivel de interfaz: Interfaz del componente, al estilo IDL (véase capítulo 2.9), es decir: Descripción de signaturas. Información puramente estructural.
- Nivel de originador: Información sobre qué tipos y versiones de componentes pueden funcionar en conjunción.
- Nivel semántico: Descripción completa de la funcionalidad de un componente.

En [SLMH96] Steyaert, Lucas, Mens y D'Hondt introducen los contratos de reutilización como un medio de descripción de interfaces de cara a la composición, intentando llenar los huecos existentes en los niveles mencionados en el párrafo anterior. En [HLS97, Lu97] proponen los contratos de reutilización no sólo como una mejor descripción de las interfaces, sino como un medio para afrontar el problema de validar la composición cuando los componentes evolucionan.

Para el PTL-VUB un contrato de reutilización no es un contrato que tiene vigor (como lo era para Meyer) para un método ni para una sola clase, sino que es una **descripción de interfaces para un conjunto de componentes que colaboran**. Describe los papeles que juegan los distintos participantes, sus interfaces, sus relaciones, y la interacción entre ellos.

La forma de expresión del contrato se hereda de Lamping [La93] pero se amplía y se le añade un formato gráfico, bastante más legible. Estos contratos expresan qué papeles participan en la interacción, entre qué participantes existe una relación, y lo que es más importante, dada una invocación a un método de un participante, queda descrito a qué métodos (y de qué participantes) llamará a su vez. Esto no implica que se describan *todas* las

llamadas, lo que equivaldría a convertir los componentes en "cajas blancas" y violar su encapsulamiento. Las llamadas que quedan documentadas son las que forman parte del contrato; de este modo, los participantes pueden asumir que los componentes con los que colaboran realizarán a su vez determinadas llamadas. Gracias a ello, la interacción entre componentes está mucho más controlada y documentada.

Al desarrollar nuevos componentes que ejerzan un determinado papel, no es imprescindible ceñirse totalmente al contrato, pero si no se hace así, el desarrollador del nuevo componente debe documentar la nueva versión del contrato con las modificaciones oportunas. Y aquí entra en juego un nuevo concepto, los **operadores de reutilización** (por ejemplo, *extensión* y *refinamiento*). Un contrato de reutilización se convierte en otro mediante el uso de un operador.

Dado que tanto los operadores válidos como sus efectos son conocidos, a la hora de ensamblar componentes nuevos es posible verificar de forma automática y temprana si las suposiciones básicas se siguen cumpliendo o no, simplemente procesando los contratos de reutilización, viendo qué operadores se les han aplicado y analizando la coherencia del resultado de forma casi matemática.

Un problema potencial de este enfoque es que un componente que declara ajustarse a un contrato de reutilización puede no hacerlo. Es decir, puede que en respuesta a un mensaje un contrato de reutilización exija realizar determinada llamada a un método, por ejemplo, y que el componente no la realice. Pero el problema de adecuación entre el código fuente y las declaraciones sobre el comportamiento del mismo afecta casi a cualquier estructura paralela al propio código, ya sean contratos de reutilización, documentos de análisis o diseño, referencias de funciones, especificaciones formales, etc. etc. Este problema es además de muy difícil solución en la práctica, aun cuando en este punto sí puedan ayudar ciertos métodos que se describen en otros puntos de este documento (métodos formales, técnicas de interpretación abstracta, o sobre todo esquemas de generación de código fuente a partir de especificaciones).

En este capítulo de revisión se ofrece sólo esta visión general de los contratos de reutilización, pero puede verse una descripción mucho más detallada sobre los mismos en el capítulo 5.2.2, como introducción al problema práctico que allí se presenta.

### 2.6.4. Lenguaje Contract (Northeastern University)

En el College of Computer Science de la Northeastern University (en adelante CCS-NEU) existe un grupo de interés (el Demeter Research Group), dirigido por Karl J. Lieberherr, que ha realizado diversas aportaciones en el campo de la orientación a objetos [LHR88], de las que quizás la más importante sea la formulación de la llamada Ley de Demeter [Ho92] pero que también ha realizado notables avances en el terreno de los componentes. En concreto, este grupo está especialmente interesado en cuestiones de reutilización.

Como medio para afrontar los problemas de reutilización incorrecta, presentan un lenguaje, *Contract*, que sirve para "representar y reutilizar componentes de grano grueso, donde el adjetivo 'de grano grueso' implica que los componentes contienen dos o más definiciones de clase y posiblemente algunas declaraciones de objetos" [Ho92, págs. 2-3]. El origen de Contract se encuentra en experiencias de reutilización de grandes bibliotecas como InterViews [LVC89]. InterViews es una biblioteca para la creación de interfaces gráficas de usuario (IGUs), y su arquitectura es del tipo "dirigida por eventos". Esto hace que sea muy

difícil comprender su funcionamiento, tener claras las complejas interacciones entre sus elementos, la ordenación temporal de los eventos, y por supuesto cómo adaptar la biblioteca o derivar nuevas clases (todo esto probablemente lo experimentará cualquiera que trabaje en un entorno de ventanas con una biblioteca de clases como ObjectWindows™ de Borland o MFC™ de Microsoft). Una función determinada del sistema rara vez estaba comprendida en una sola clase, sino que surgía de la colaboración entre varias clases, y por tanto la responsabilidad de un comportamiento concreto estaba dispersa. Con objeto de clarificar y hacer explícitas estas complejas relaciones entre clases, comenzó el trabajo de desarrollar una notación adecuada, trabajo que culminó en el lenguaje Contract.

En este lenguaje, la entidad principal es precisamente el contrato, que en este ámbito es un paquete de declaraciones de objetos y definiciones de comportamiento de dichos objetos, limitados por un ámbito de visibilidad.

La definición de contrato, por tanto, difiere de las de Meyer, Wirfs-Brock et al y se acerca más a la interpretación del PTL-VUB; el contrato implica a varias clases y varios métodos de las mismas, y refleja la interacción entre ellas en múltiples sentidos, describiendo incluso el ordenamiento temporal de las llamadas. En palabras de sus autores, el lenguaje Contract es una extensión del modelo habitual de programación orientada a objetos implementada por los principales lenguajes orientados a objetos, como C++, Eiffel o Smalltalk. El concepto central del lenguaje es la *interacción entre objetos*, que hace referencia a un grupo de objetos que interaccionan entre sí mediante el paso de mensajes para realizar alguna tarea del sistema o cooperar a mantener un invariante del mismo. En esa interacción, cada objeto proporciona cierta funcionalidad, a la vez que confía en otros objetos para que proporcionen el resto del comportamiento deseado.

En algunos aspectos, este enfoque del concepto de *contrato* se acerca al propuesto en esta tesis; Holland recoge una idea fundamental expresada por Biggerstaff y Richter [BR87], y es que "los diseños deben representarse en una forma que pueda ser procesada por una máquina". Este planteamiento es el que motiva los requisitos sobre automatización que ya se han expresado en este trabajo, y es que cuando se habla de *verificación* se hace referencia a un proceso automatizable y no al simple criterio humano sobre corrección (aunque, como se discutirá en su momento, la propuesta de CCS-NEU tiene poca relación con el tipo de verificación que aquí se persigue).

Un contrato, según Holland, tiene seis partes:

1. Un **nombre de contrato**.
2. Una **cláusula de refinamiento**. Un contrato puede definirse como el refinamiento de otro contrato previamente existente.
3. Una **lista de participantes**. Cada contrato declara una lista de objetos que son los participantes de ese contrato. Para cada participante, se denota también cuál es su *obligación* dentro del contrato.
4. **Definiciones de obligaciones**. Se detalla en qué consiste cada *obligación* de las mencionadas en el punto anterior. La definición de obligación se parece (sintácticamente) a la definición de una clase: contiene variables de instancia, métodos (incluyendo la implementación de los mismos, utilizando en tal

implementación invocaciones a los demás participantes del contrato) e interfaces de métodos (en el caso de que no se suministre implementación alguna).

5. **Cláusulas de inclusión**.
6. **Definiciones de invariantes**.

El lenguaje Contract pretende llenar el vacío existente en la representación textual utilizada para describir interacciones en software orientado a objetos.

¿Cómo *utilizar* un contrato? Cuando una serie de clases se van a integrar para cumplir el contrato descrito, el primer paso son las **cláusulas de adecuación** (*conformance clauses*). Si se tiene un contrato en el que hay un participante `Cliente` y un participante `Servidor`, y se desea utilizar la clase `Browser` y la clase `ServidorHTTP` de forma que sigan el modelo de interacción descrito en dicho contrato, la primera operación consiste en declarar que la clase `Browser` cumple el papel `Cliente`, y la clase `ServidorHTTP` cumple el papel `Servidor`. Lo mismo vale para las operaciones implicadas (métodos); se establece un mapeo entre las clases / métodos (elementos "reales" de la implementación) y los papeles / métodos (elementos que formaban parte del contrato). De este modo, para las operaciones del contrato pueden utilizarse varias implementaciones diferentes, y lo mismo ocurre con los papeles o participantes.

Las cláusulas de adecuación se utilizan, pues, en el plano de las *clases*. Llegado el momento de la implementación definitiva, se puede *poner a funcionar* un contrato asignando instancias de las clases (es decir, objetos) a los diversos papeles. Este proceso recibe el nombre de **instanciación**. Exige que en la implementación se use una construcción específica, un objeto especial llamado una *lente de contrato* que almacena las referencias a los participantes y decide el significado de las operaciones. Por supuesto, la lente de contrato actúa de acuerdo con la cláusula de adecuación que se le haya suministrado para su creación.

Así, el mismo contrato puede cumplirse entre diferentes clases y métodos (mediante diferentes cláusulas de adecuación) y posteriormente con diferentes instancias de las clases (mediante diferentes instancias de las lentes de contrato, que a su vez se apoyan en alguna cláusula de adecuación).

El uso de Contract introduce nuevos elementos en un programa, puesto que en la implementación existe la figura de las "lentes de contrato" (*contract lenses*) que son clases que juegan siempre un papel de "intermediario" en las invocaciones entre métodos de los participantes en un contrato. Además, los contratos se activan y desactivan; cada clase puede participar en varios contratos a la vez (las clases tienen "múltiples interfaces", entendiendo la interfaz como un subconjunto de sus métodos que, mediante una sentencia de adecuación, se hace corresponder con cierto papel en cierto contrato).

El lenguaje Contract presenta como características principales las siguientes:

- Da soporte a la representación sintáctica explícita de interacciones entre objetos.
- Proporciona un ente de tipo "módulo" (el contrato) para estructurar y organizar implementaciones orientadas a objetos.
- Los componentes definidos en los contratos pueden adaptarse mediante sustitución de tipos y redeclaración de métodos, y de este modo los contratos pueden describir fragmentos reutilizables de programas (también llamados *esquemas parcialmente interpretados*).

- Da pie a reutilizar unos contratos en otros (gracias a las sentencias de inclusión y de refinamiento).

# 2.7. Estilos arquitectónicos e incoherencias

## 2.7.1. Definición de estilos arquitectónicos: Carnegie Mellon (CSSRG-CMU)

Los trabajos sobre componentes de software a título genérico se remontan a algunos años atrás [BS92] y el interés general hacia los mismos se extendió durante la década de los 90. Uno de los grupos que ha recogido de forma explícita el hecho de que la combinación de sistemas puede traer problemas nuevos que no se derivan de ningún defecto en ninguno de los sistemas componentes, sino de la combinación misma, es el formado en la School of Computer Science de la Universidad Carnegie Mellon, denominado Composable Software Systems Research Group (y que aquí denominaremos CSSRG-CMU). Aun antes de adoptar ese nombre, miembros de este grupo (como Jeannette Mary Wing) han realizado aportaciones en diversas áreas de la especificación formal de componentes [Wi95, ZW97], pero en esta disertación se entenderá que la mención al CSSRG-CMU hace referencia al trabajo sobre estilos arquitectónicos. En este grupo de investigación se incluye David Garlan, frecuente referencia en los trabajos de otros investigadores. Con Mary Shaw ha publicado trabajos que sientan las bases del concepto de "arquitectura software" [SG93, SG96]. Junto con Robert Allen y John Ockerbloom, es coautor de un artículo que también debe considerarse seminal en este campo [GAO95]. Yendo más allá de la mera definición de lo que es la arquitectura software, en él se exponen con toda claridad los fundamentos del problema de la composición, y se explica (y esto está extraído literalmente del título) "por qué es difícil construir sistemas uniendo partes ya existentes".

Según este texto, actualmente se piensa que las mejoras drásticas en productividad en el desarrollo de software dependerán de nuestra capacidad para combinar piezas de software ya existentes con el fin de formar nuevas aplicaciones. Los autores formulan, como consecuencia, la necesidad de desarrollar nuevas técnicas para detectar y afrontar incongruencias entre los elementos ensamblados, lo que coincide plenamente con los planteamientos de esta tesis. No obstante, aunque este grupo admite que muchos problemas tienen origen en la interoperabilidad a bajo nivel, centra su interés en otro tipo de problemas, las incoherencias arquitectónicas. A juicio de los autores, estos problemas de alto nivel pueden ser tan graves e incluso tan difíciles de detectar como los de bajo nivel.

Otras dificultades para la formación de sistemas complejos a partir de componentes son:

- Falta de componentes a partir de los cuales construir.
- Imposibilidad de encontrar los componentes aun cuando de hecho existen.

Evidentemente, hay líneas de investigación abiertas en ambos campos, con el desarrollo de bibliotecas y de técnicas para localizar componentes apropiados [ZW98]. Pero dejando esto aparte, se pone de relieve que muchos problemas surgen simplemente porque los componentes *no encajan bien*.

Las publicaciones de este grupo reflejando sus experiencias al intentar combinar diversos componentes software (de grano muy grueso) y los principales problemas encontrados

sentaron una base terminológica y de enfoque sobre la que han evolucionado posteriores trabajos. En gran parte, su propuesta se basa en la idea de que se debe tener presente el *estilo arquitectónico* empleado, e intentar limitarse a unir sistemas que sean coherentes entre sí en este aspecto (intentar combinar sistemas del mismo estilo para evitar los problemas de composición).

## 2.7.2. Lenguajes de Descripción de Arquitecturas (ADL)

ABLE [ABLE] es un grupo de investigación de la universidad Carnegie-Mellon, relacionado con el CSSRG-CMU (de hecho, Garlan, Allen y Ockerbloom forman parte de ABLE). El proyecto ABLE pretende explorar las bases formales de la arquitectura software y construir herramientas que puedan ser útiles para los arquitectos de software. El primer objetivo se aborda en el lenguaje WRIGHT, mientras que el segundo ha dado lugar al sistema Aesop. Posteriormente se ofrecerá una breve visión de estos proyectos.

Los Lenguajes de Descripción de Arquitecturas, en adelante ADLs (*Architecture Description Languages*) proporcionan notaciones para descomponer un sistema en componentes y conectores y especificar cómo se combinan estos elementos para formar cierta configuración. Los métodos formales (véase 2.2) presentan el problema de que, debido a su generalidad, al aplicarlos a la descripción de un estilo arquitectónico es necesario realizar una tarea costosa y no trivial, que es definir los elementos de la arquitectura a partir de las primitivas (normalmente de muy bajo nivel) de la notación formal que se utiliza. Esto suele tener como consecuencia que se realicen, como mucho, descripciones formales parciales y *ad hoc*. Un ADL tiene la ventaja de que ofrece una notación descriptiva directa para las principales abstracciones arquitectónicas (fundamentalmente, para los llamados *componentes* y *conectores*, que se describen más adelante). De este modo, los arquitectos pueden exponer y definir la estructura de sus sistemas, razonar sobre ella, evaluar posibles cambios, guiar la implementación, etc. evitando la ambigüedad.

Existen, además, ADLs específicos, que cuentan con entornos de desarrollo asociados a su propósito; por ejemplo, la notación MetaH proporciona un entorno completo para el desarrollo e implementación de sistemas de control aeronáutico. Sin embargo, salvo algunas excepciones, los ADLs no suelen proporcionar una forma directa de especificar propiedades de elementos individuales, como el patrón de interacción de cierto conector. Esto limita el proceso analítico que puede realizarse sobre dichas notaciones [Al97, p. 7].

### WRIGHT

El lenguaje WRIGHT [Al97] intenta aunar lo mejor de los dos enfoques mencionados en 2.7.2. Los métodos formales son demasiado generales, pero ofrecen un gran margen para la deducción analítica. Los ADLs son más útiles en el mundo real por ser utilizables de forma más directa, pero esa especificidad a veces es una traba, y en muchas ocasiones no resultan muy útiles para realizar procesos analíticos sobre ellos. WRIGHT es precisamente un ADL con una sólida base formal.

WRIGHT se basa en la descripción formal del comportamiento abstracto de los componentes y conectores. Modela los diversos tipos de conector como patrones abstractos de interacción mediante una notación inspirada en CSP (véase 2.5.1) y define los estilos arquitectónicos como predicados que restringen las posibles configuraciones. Los conectores definen un modelo abstracto de la relación entre acciones discretas y asíncronas

de los componentes, y estos modelos de patrones de interacción son independientes de las instancias de componente concretas.

Para la descripción y análisis de una arquitectura, según Allen la notación o modelo elegido debe dar soporte a:

- La descripción de configuraciones arquitectónicas. En otras palabras, qué componentes intervienen en un sistema y qué interacción hay entre ellos.
- La descripción de estilos arquitectónicos. Más allá de la descripción de sistemas concretos, debe ser posible hablar de familias de sistemas, estilos que aprovechan los aspectos comunes entre los sistemas para facilitar el análisis e implementación.
- El análisis de propiedades interesantes. La descripción del sistema debe poder utilizarse para, mediante el análisis adecuado, determinar si el sistema se ajusta a los requisitos planteados.
- La aplicación a problemas prácticos en sistemas reales. Una notación práctica debe poder ser utilizada en sistemas complejos, reales.

La descripción de un componente en WRIGHT consta de dos apartados: la *interfaz* y la *computación*. En WRIGHT, la interfaz consta de cierto número de *puertos*, y cada puerto define una de las interacciones (independientes) en las que el componente puede participar. El puerto define algunos aspectos del comportamiento del componente (los relacionados directamente con tal puerto); en cierta manera, es una *especificación parcial* del componente.

Un conector representa una interacción entre varios componentes. La descripción de un conector consta a su vez de los *papeles* y el *pegamento*[1]. Cada papel define el comportamiento de un participante en la interacción. Un *pipe* tiene dos papeles, el emisor de datos y el receptor. El pegamento del conector define cómo interactúan entre sí los diversos papeles.

Cada una de las partes de una descripción (los puertos, papeles, computaciones y pegamento) se definen utilizando una variante de CSP (véase capítulo 2.5.1).

El uso de notaciones precisas como WRIGHT facilita la comunicación de ideas entre arquitectos. Además, proporciona una base para analizar las arquitecturas. Se pueden usar las descripciones de conectores para determinar si el conector satisface ciertas propiedades críticas, como la consistencia interna del protocolo y si los papeles están lo suficientemente restringidos como para garantizar un comportamiento correcto de los componentes. Por supuesto, las descripciones de componentes y conectores juegan el papel de "tipos" en el sentido de que son generales y abstractas, y en el sistema final intervendrán instancias concretas de los mismos.

Otro aspecto importante de WRIGHT es su capacidad para describir estilos arquitectónicos. Una descripción de un estilo consta de dos partes: vocabulario y restricciones sobre las configuraciones. El vocabulario se construye mediante la simple definición de tipos de componente y conector, mientras que las restricciones son predicados que deben cumplirse

[1] Traducción literal del término *glue*; en otros puntos de este documento se utiliza la expresión *código de unión*, pero en este caso no se está haciendo referencia necesariamente a código. Por otra parte, se ha evitado traducir *glue* por *cola* debido a que en informática este término ya tiene asociados otros significados que pueden mover a confusión.

en cualquier sistema del estilo en cuestión. Basándose en descripciones de estilos, la posterior descripción de un sistema es mucho más simple, ya que no es necesario repetir la información contenida en la descripción del estilo.

Las configuraciones de sistema en WRIGHT son jerárquicas; cada componente de un sistema puede describirse mediante una especificación de componente propiamente dicha o mediante una nueva descripción de sistema.

### Aesop

Aesop [Aesop, Gr95] es una herramienta que permite generar entornos de diseño arquitectónico. Para ello, el usuario suministra a Aesop descripciones de estilos arquitectónicos, lo que incluye el vocabulario de elementos de diseño (componentes, conectores, patrones) junto con su semántica asociada, reglas globales de diseño, etc. A partir de esa información, Aesop genera el mencionado entorno de diseño, al que además pueden añadirse diversas herramientas específicas, tales como detectores de ciclos, sistemas de verificación de consistencia de tipos, analizadores formales de protocolos de comunicación, generadores de código, etc. Por ejemplo, la Ilustración 5 muestra un entorno de diseño para sistemas que se encuadran en el estilo arquitectónico *tubería-y-filtro*. A la derecha figuran los elementos que el usuario puede seleccionar para crear instancias suyas en el sistema.

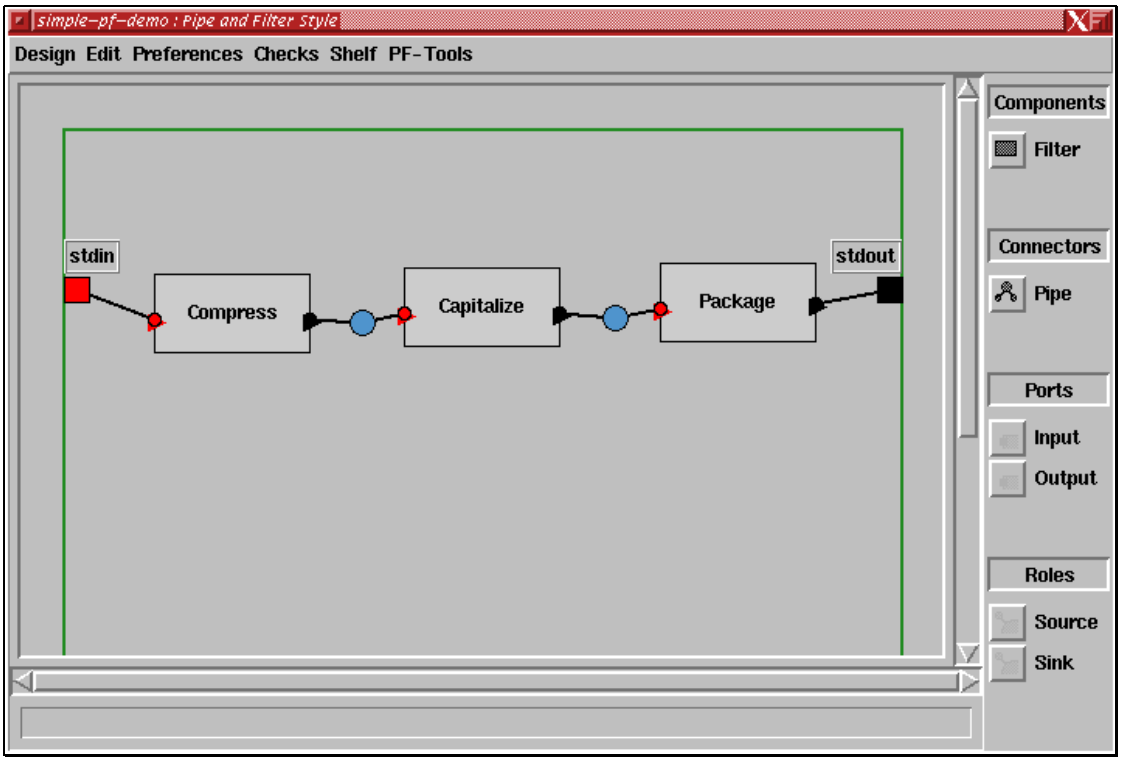



*Ilustración 5. Un entorno generado por Aesop en acción.*

En términos generales, un entorno creado con Aesop ofrece:

1. Una paleta de tipos de elementos de diseño que se corresponden con el vocabulario del estilo arquitectónico en cuestión.

2. Verificaciones de que las composiciones de estos elementos satisfacen las restricciones topológicas de ese estilo.
3. Especificaciones semánticas opcionales de los elementos.
4. Una interfaz que permite que herramientas externas analicen y manipulen las descripciones arquitectónicas.
5. Diversas visualizaciones (propias de cada estilo) de la información arquitectónica, junto con uno o más editores gráficos para su manipulación.
6. Una base de datos de diseño, en la que se almacenan (y de la que se recuperan) los diseños.
7. Una o más bibliotecas de reutilización de arquitecturas, para almacenar y recuperar fragmentos de diseño ya existentes.

**Darwin**

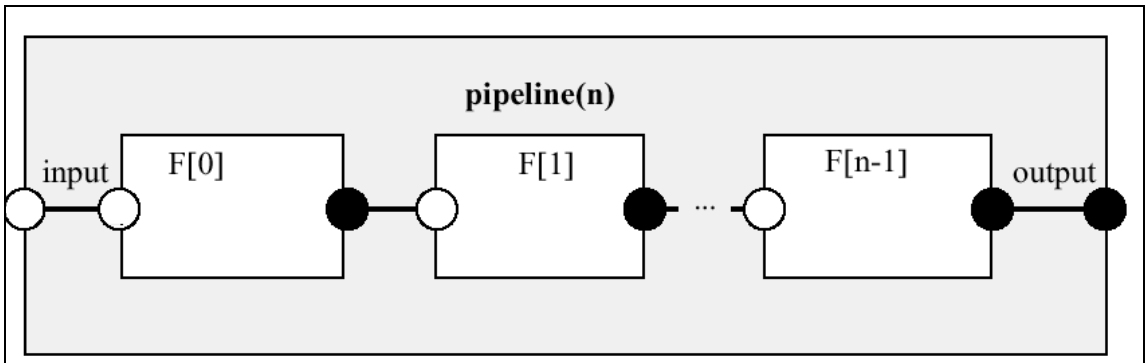
Darwin [MDEK95] es un ADL desarrollado en el Imperial College, cuyo objetivo principal es estructurar los complejos patrones de interconexión que se producen al desarrollar un sistema distribuido y concurrente. La interpretación distribuida y concurrente de Darwin es una de sus principales características.

Como en otros ADLs, existe el concepto de que los componentes proporcionan y requieren servicios; además, estos servicios son *locales* a la especificación del componente, en el sentido de que un componente no necesita conocer los nombres globales de servicios externos. Cada componente puede, pues, implementarse y probarse de manera independiente del sistema del que forman parte. Esta propiedad se denomina *independencia del contexto*. La descripción de un componente se puede reutilizar al definir un sistema, instanciándolo repetidas veces.

Para describir un sistema, Darwin ofrece una notación textual y gráfica. Los servicios ofrecidos se representan mediante círculos rellenos, y los requeridos mediante círculos huecos. En la Ilustración 6 puede verse un ejemplo de componente que a su vez está compuesto de otros, expresado con la notación gráfica y textual de Darwin. La expresión `@k+1` indica que cada componente estará alojado en una máquina diferente.

El resultado de la descripción de un sistema en Darwin es un componente compuesto, estructurado jerárquicamente, que en tiempo de ejecución dará lugar a una serie de instancias de componentes, concurrentes y potencialmente distribuidos.

Para describir un sistema, es necesario poner en relación servicios ofrecidos por un componente con los servicios requeridos por otros componentes, lo que se consigue mediante la sentencia `bind`. Este enlace puede producir problemas; los servicios ofrecidos y requeridos deben ser de **tipos compatibles**. Dicha compatibilidad la determina la plataforma de sistemas distribuidos de destino (es decir, en la que se ejecutará el sistema resultante). Las herramientas de Darwin pueden inferir el tipo de un servicio que no se ha hecho constar explícitamente.




```
component pipeline(int n) {
 provide output;
 require input;

 array F[n]: filter;
 forall k:0..n-1 {
   inst F[k] @ k+1;
      when k < n1;
     bind F[k+1].input -- F[k].output;
 }
 bind
   F[0].input -- input;
   output -- F[n-1].output;
}
```

*Ilustración 6. Un sistema de ejemplo en Darwin.*

En general, cada servicio ofrecido puede enlazarse con muchos servicios requeridos; cada servicio requerido, sin embargo, se enlaza con un solo servicio ofrecido.

Es posible (y así se hace en [MDEK95]) modelar Darwin mediante el π-cálculo (véase 2.5.2), lo que permite a sus autores ofrecer una semántica precisa para el lenguaje.

**Rapide**

Rapide [Rapide, Lc96, Lc98] es un proyecto del Program Analysis and Validation Group (PAVG) de la Universidad de Stanford. Pretende ayudar en el desarrollo de sistemas distribuidos, multilenguaje y de gran tamaño. Constituye un caso de Lenguaje de Definición de Arquitecturas Ejecutable (EADL) para el desarrollo evolutivo y análisis riguroso de grandes sistemas. Este proyecto nace sobre la base de trabajo que el PAVG había dedicado a la aplicación de métodos formales a Ada y VHDL durante los setenta y los ochenta.

El lenguaje se ha diseñado para dar soporte al desarrollo basado en componentes de sistemas multilenguaje de gran tamaño, utilizando las definiciones arquitectónicas como marco fundamental. Las descripciones arquitectónicas se van refinando gradualmente en productos, y permite realizar pruebas y mantenimiento basándose en la comparación (automatizada) con arquitecturas formales estándar. El modelo de eventos adoptado por

Rapide proporciona la base formal para sus herramientas de prototipado y mantenimiento, con el fin de realizar análisis de corrección y eficiencia en sistemas distribuidos.

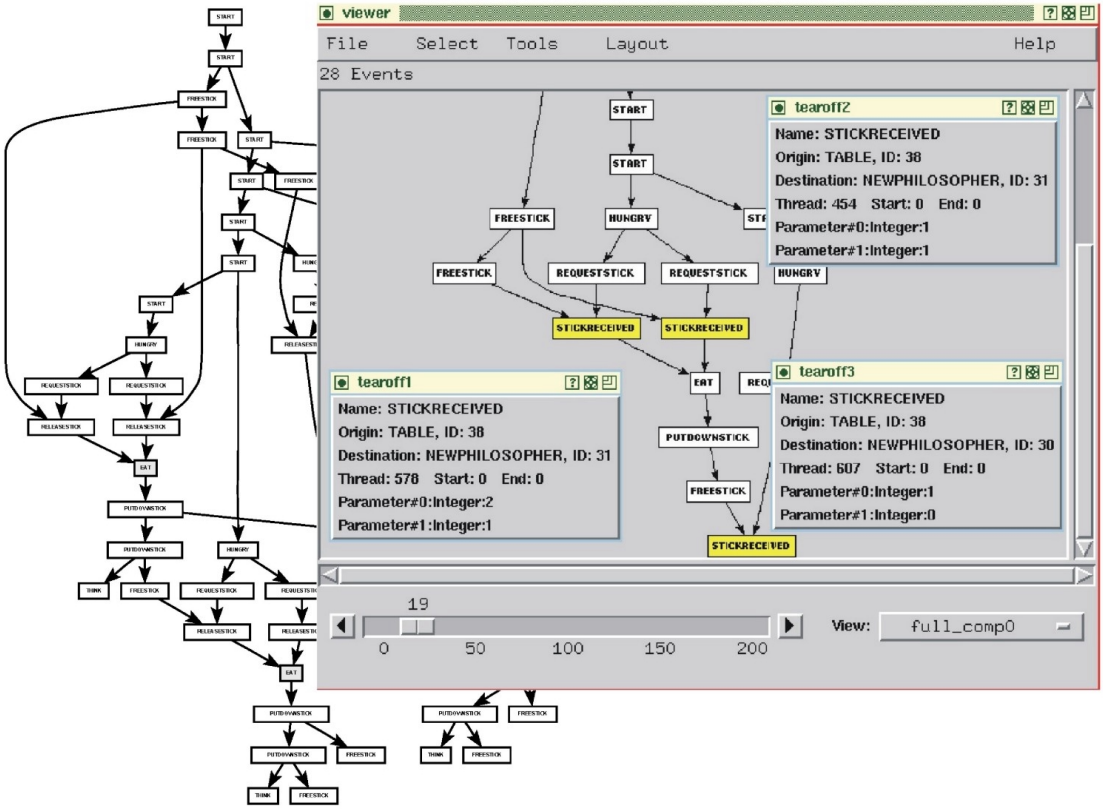



*Ilustración 7. Poset de "la cena de los filósofos" y su evaluación en POV.*

El lenguaje de especificación permite definir restricciones sobre patrones de eventos concurrentes (*posets*, de *Partially Ordered SETs*, "conjuntos parcialmente ordenados") y también restricciones de tiempo real. (En la Ilustración 7 puede verse el grafo de un poset, el del modelo del típico problema de concurrencia, "la cena de los filósofos", y su evaluación con la herramienta POV). Con las herramientas asociadas, se puede realizar un análisis formal automatizado (mediante demostraciones teóricas y verificaciones de simulación) de propiedades muy concretas de los sistemas distribuidos, que incluyen tanto hardware como software. En realidad, Rapide contiene varios sublenguajes:

- Un lenguaje de tipos.
- Un lenguaje de definición de arquitecturas ejecutable.
- Un lenguaje de especificación.
- Un lenguaje de programación reactiva concurrente.

El **lenguaje de tipos** se basa en una estructura general de interfaz (en el sentido clásico del término), junto con ciertas derivaciones para construir nuevas interfaces a partir de las existentes. Esta "derivación" se parece a la herencia de los lenguajes orientados a objetos.

El lenguaje de tipos se aplica, pues, a la definición de las interfaces de los componentes (ya sean hardware o software) de un sistema.

El **lenguaje de definición de arquitectura** proporciona características ejecutables para componer sistemas a partir de las interfaces de los componentes; basta con definir sus interconexiones de sincronización y comunicación en términos de patrones de eventos. Existen, además, mecanismos para utilizar los patrones de eventos en la definición de correspondencias entre diferentes arquitecturas, lo que abre la puerta a la implementación de herramientas que comparen las arquitecturas y verifiquen, por ejemplo, su grado de conformidad con estándares.

El **lenguaje de restricciones** permite realizar una especificación abstracta del comportamiento de un sistema distribuido, incluyendo requisitos temporales. Es un lenguaje basado en el modelo *poset*, que permite definir patrones de eventos que se requieren o prohíben en cierto sistema distribuido.

El **lenguaje ejecutable** es un lenguaje de programación reactiva concurrente. Utiliza tipos, objetos y expresiones del lenguaje de tipos, y proporciona estructuras de control y de módulos. Sus principales entidades son procesos reactivos independientes que se activan cuando durante la ejecución se producen ciertos patrones de eventos. Estos procesos activados por patrones se usan para:

1. Definir conexiones arquitectónicas entre los componentes.
2. Construir comportamientos de los componentes mediante técnicas de programación reactiva basada en reglas.

El lenguaje ejecutable también ofrece estructuras de control, subprogramas, manejo de excepciones y características de temporización.

### UniCon

UniCon [UniCon, SDK95] es un ADL creado (bajo la influencia del mismo personal involucrado en CSSRG-CMU, como Mary Shaw o David Garlan) para dar soporte a diversos estilos y elementos arquitectónicos, y para construir sistemas a partir de sus descripciones arquitectónicas. El nombre UniCon proviene de la expresión Universal Connector.

La descripción de una arquitectura en UniCon consta de un conjunto de componentes y conectores. Un componente es una entidad que contiene datos y / o computaciones, y un conector es un mediador que regula la interacción entre componentes. Cada componente tiene una interfaz que exporta un conjunto de *actores*[2], equivalentes a los *roles* en otras terminologías. Estos actores representan las maneras en las que el componente puede interactuar con el exterior. Análogamente, el protocolo de un conector exporta unos *papeles* que representan las maneras en las que el conector es capaz de actuar como mediador en una interacción. Puede verse un ejemplo gráfico de esto en la Ilustración 8.

[2] *Players* en el original.

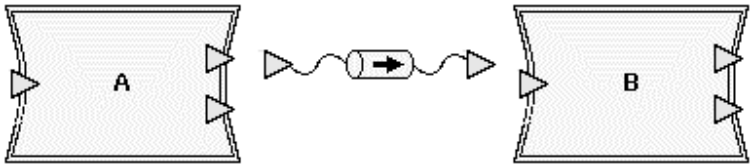



*Ilustración 8. Diagrama de ejemplo producido con el editor gráfico de UniCon.*

En este diagrama de ejemplo aparecen dos componentes, A y B, que son filtros Unix. Cada uno de ellos exporta tres actores, que aparecen representados como triángulos; el actor de la izquierda en cada componente representa la entrada estándar, y los actores de la derecha representan la salida estándar y la salida estándar de errores (stdin, sdout y stderr respectivamente, en terminología Unix). Entre los dos componentes aparece un conector, que representa una tubería (*pipe*) de Unix, que ofrece dos papeles: el de su izquierda sería la entrada de datos en la tubería, y el de la derecha la salida de los datos de la tubería.

En un editor gráfico, estos elementos (los componentes y la tubería) se conectarían, probablemente mediante una simple acción de *arrastrar y soltar*, y mediante ese procedimiento se asocian los actores con sus papeles, lo que da lugar a la configuración deseada para el sistema.

En principio, existe un conjunto de conectores posibles, que puede crecer en el futuro. Asimismo, el modelo define una serie de componentes predeterminados, describe qué conexiones son posibles entre componentes y conectores, qué atributos tienen los componentes, qué tipos de conectores existen, etc.

No es objeto de este texto describir en detalle cada uno de estos elementos, sino sólo hacer notar el hecho de que están claramente enumerados y establecidos (en algunos casos, su significado puede deducirse con facilidad de su nombre). Como orientación, un resumen de los componentes predefinidos puede verse en la Tabla 1. Los atributos asociados a los componentes son InstFormals, Variant, RecordFormat, Library, EntryPoint, Priority, Processor, SegmentDef, TriggerDef, RPCTypesIn y RPCTypedef. Asimismo, los actores también tienen sus atributos: MaxAssocs, MinAssocs, Signature, PortBinding, Member, SegmentSet y Trigger. En la Tabla 2 pueden verse los tipos de conectores predefinidos y los papeles (*roles*) que ofrecen. Los atributos de los conectores son InstFormals, PipeType, Match, Algorithm, Processor y Trace, y los atributos de los papeles o roles son Accept, MaxConns y MinConns.

| Componentes | Tipos de actores soportados |
|---|---|
| Module | RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, PLBundle, ReadFile, WriteFile |
| Computation | RoutineDef, RoutineCall, GlobalDataUse, PLBundle |
| SharedData | GlobalDataDef, GlobalDataUse, PLBundle |
| SeqFile | ReadNext, WriteNext |
| Filter | StreamIn, StreamOut |
| Process | RPCDef, RPCCall |
| SchedProcess | RPCDef, RPCCall, RTLoad |
| General | Todos (se permite cualquier tipo de actor) |

*Tabla 1. Componentes predefinidos en UniCon y actores posibles.*

| Tipo de conector | Papeles y actores aceptables |
|---|---|
| Pipe | Source (acepta StreamOut de Filter, ReadNext de SeqFile)<br>Sink (acepta StreamIn de Filter, WriteNext de SeqFile) |
| FileIO | Reader (acepta ReadFile de Module)<br>Readee (acepta ReadNext de SeqFile)<br>Writer (acepta WriteFile de Module)<br>Writee (acepta WriteNext de SeqFile) |
| ProcedureCall | Definer (acepta RoutineDef de Computation o Module)<br>Caller (acepta RoutineCall de Computation o Module) |
| DataAccess | Definer (acepta GlobalDataDef de SharedData o Module)<br>User (acepta GlobalDataUse de SharedData, Computation, o Module) |
| PLBundler | Participant (acepta PLBundle, RoutineDef, RoutineCall, GlobalDataUse, GlobalDataDef de Computation, Module o SharedData) |
| RemoteProcCall | Definer (acepta RPCDef de Process o SchedProcess)<br>Caller (acepta RPCCall de Process o SchedProcess) |
| RTScheduler | Load (acepta RTLoad de SchedProcess) |

*Tabla 2. Conectores predefinidos en UniCon, papeles y actores aceptados.*

Según sus autores, el uso de estructuras de composición con base formal (como en el caso de UniCon) tendrá efectos beneficiosos en la producción de software:

1. Permitir la elección de paradigmas de diseño que se ajusten a las características deseadas en los sistemas.
2. Permitir el desarrollo de marcos y arquitecturas de referencia de dominio específico.
3. Habilitar técnicas de desarrollo que exploten las propiedades de los sistemas como compuestos.
4. Dar soporte a un alto grado de parametrización, de modo que los sistemas de gran tamaño se puedan diseñar, comprender, mantener y mejorar con mayor facilidad.
5. Habilitar la reutilización de código preexistente y proporcionar vías para recuperar información arquitectónica parcial de los sistemas existentes.

**Otros**

Existen muchos otros ADLs y sistemas derivados. En [ADLTK] puede verse una relación en la que aparecen muchos de ellos. Como representantes de cierta entidad cabe citar ACME (un ADL simple y genérico utilizado con frecuencia como formato de intercambio entre otros ADLs), Maude (que no es un ADL sino un sistema basado en reescritura pero que guarda cierta relación con los ADLs), RESOLVE, etc.

No es objeto de esta tesis un análisis exhaustivo de los ADLs, sino sólo un recorrido por los más representativos, que permita sopesar hasta qué punto responden al planteamiento de esta tesis (lo que se hará en un capítulo posterior).

### 2.7.3. Combinación de estilos arquitectónicos: Southern California (CSE-USC)

Otro grupo muy activo en el campo de la combinación de componentes software es el Centro para la Ingeniería del Software de la Universidad de California del Sur, o CSE-USC (*Center for Software Engineering, University of Southern California*). Cuenta con la participación de autores de renombre como Barry Boehm, y otros (Cristina Gacek, Ahmed Abd-Allah, Nenad Medvidovic) que han publicado trabajos que son también referencia obligada en el campo de la arquitectura software. Bebiendo de las fuentes del CSSRG-CMU, este grupo ha realizado una profunda exploración de los estilos arquitectónicos existentes, y ha entrado en detalles sobre la figura de las **características conceptuales** (*conceptual features*) que constituyen prácticamente la definición de los estilos arquitectónicos.

Cada estilo arquitectónico (y se recoge cerca de una decena de ellos) posee unas determinadas características conceptuales que lo identifican. Al combinar sistemas que se inscriben en estilos arquitectónicos diferentes, surgen estilos arquitectónicos heterogéneos, pero no todas estas combinaciones son viables. El CSE-USC ha abordado la tarea de la formulación de dichos estilos [GACB95, Ab96] y de las características conceptuales que les son propios, y ha investigado sobre la viabilidad de dichas composiciones [AB95, Ga97]. El trabajo en este campo ha continuado en nuevas líneas; Cristina Gacek ha dirigido investigaciones destinadas a incluir nuevos estilos arquitectónicos en el esquema general, y a encontrar problemas y soluciones en la combinación de estos nuevos estilos con los que ya estaban identificados.

Por tanto, el CSE-USC, recogiendo las recomendaciones del CSSRG-CMU sobre la conveniencia de tener presentes los estilos arquitectónicos y los potenciales problemas, ha intentado ir más allá de la mera prudencia, desgranando el problema de la composición de diferentes estilos, realizando clasificaciones e identificando fricciones concretas entre ellos. De este modo, sus investigaciones han producido:

- Una lista de estilos arquitectónicos ampliada que sirve de base terminológica común.
- Una enumeración de características conceptuales para todos estos estilos, lo que constituye un punto de partida para confrontarlos de manera más formal.
- Una lista de problemas muy concretos que surgen al combinar estos estilos (por ejemplo, en [Ab96] se identifican 16 posibles incoherencias, y [Ga97] aporta otras nuevas como fruto de la inclusión de nuevos estilos arquitectónicos).

Todo esto proporciona herramientas conceptuales sobre las que discutir los obstáculos y soluciones que surgen en la composición, incluso vías de formalización para verificar la corrección de la misma. De este modo, el CSE-USC reconoce el peligro de la composición de estilos arquitectónicos diferentes, pero afronta ese reto y acepta con toda claridad la aplicación de estilos heterogéneos.

# 2.8. Metodologías de análisis y diseño

## 2.8.1. OCL (Object Constraint Language)

El desarrollo de las metodologías de análisis y diseño orientados a objetos trajo consigo la existencia de diversas notaciones y propuestas, entre las que cabe destacar la metodología de Booch [Bo94], la *Object Modeling Technique* (OMT) de Rumbaugh et al [Ru91] y OOSE de Ivar Jacobson [Ja94] entre otras. Ante el problema de la proliferación de notaciones distintas, los autores más relevantes unieron esfuerzos para producir una notación unificada, y el resultado de esto fue el Lenguaje Unificado de Modelado o *Unified Modeling Language*, UML [Ru98, BJRR98, Ja99, UML].

En palabras de sus creadores [UML], UML es un lenguaje para especificar, visualizar, construir y documentar los artefactos de los sistemas software, así como para modelado de negocios y otros sistemas no-software. UML representa una colección de prácticas de ingeniería aconsejables que han demostrado ser útiles para modelar sistemas grandes y complejos. Fue desarrollado por Rational Software, la empresa de Grady Booch, y sus socios, y en 1997 fue remitido al Object Management Group [OMG] en lo que sería la versión 1.1 de UML. Desde su creación, UML ha sido adoptado por muchas empresas como estándar en sus procesos de desarrollo, para modelado de negocios, gestión de requisitos, análisis y diseño, programación y pruebas. Actualmente, UML es gestionado por el OMG como un estándar más.

UML incluye artefactos para modelar clases, máquinas de estados, casos de uso y otros elementos de la orientación a objetos. OCL [Ra97, WK99] forma parte de UML, y sirve para especificar restricciones, pre y postcondiciones, etc. sobre los objetos de los modelos. Fue propuesto por IBM y otros en enero de 1997.

En concreto, según IBM [OCL] OCL es:

- **Un lenguaje de expresión puro**. Esto significa que una expresión OCL se garantiza que no tiene efectos laterales. No puede cambiar nada del modelo, y por tanto el estado del sistema nunca cambiará por incluir una expresión OCL, aun cuando dicha expresión puede utilizarse para especificar tal cambio de estado. La evaluación de una expresión OCL no tiene más efecto que devolver un valor.
- **Un lenguaje de modelado**. OCL no es un lenguaje de programación. No se puede escribir lógica de control en OCL; no se pueden invocar procesos u operaciones desde OCL. Por esta razón, no está garantizado que todo lo que forma parte de OCL sea ejecutable de manera directa.
- **Un lenguaje formal**. Todas las construcciones de OCL tienen un significado definido formalmente, y la especificación de OCL forma parte de la especificación de UML. No obstante, no pretende sustituir a otros lenguajes formales como VDM o Z. El hecho de que OCL sea un lenguaje formal ayuda a escribir especificaciones no ambiguas, en contraste con las especificaciones en lenguaje natural tradicionales en el modelado orientado a objetos.
- **Fácil de usar por personal de desarrollo**. Los lenguajes formales tradicionales son difíciles de utilizar por personas que no tengan una buena base en matemáticas

[OCL] pero OCL ha sido pensado para quien típicamente se dedica al modelado de negocios o sistemas.

OCL resulta aplicable para diversos propósitos [Ra97]:

- Especificar invariantes en clases y tipos en el modelo de clases.
- Especificar invariantes de tipo para estereotipos.
- Describir pre / postcondiciones en operaciones y métodos.
- Describir *guards*.
- Como lenguaje de navegación.
- Para especificar restricciones en operaciones, del tipo `operación(param1, param2) = expresión`.

### 2.8.2. Catalysis

Catalysis [SW99] es un conjunto de métodos de desarrollo basado en componentes, desarrollado por D'Souza y Wills. Básicamente, por lo que respecta a los componentes, Catalysis proporciona una forma de definir de manera no ambigua lo que se espera de un componente u objeto individual, de manera independiente a las posibles implementaciones, así como para asegurar que las implementaciones respetan sus respectivas especificaciones.

Se trata de un conjunto de métodos o patrones de actuación, especialmente para proyectos grandes. No postula un proceso de desarrollo rígido, sino que ofrece diferentes pautas y técnicas que en cada proyecto se elegirán y aplicarán de forma adecuada a los fines concretos del mismo.

Catalysis proviene de diversas fuentes. El trabajo de sus autores con métodos rigurosos de desarrollo orientado a objetos comenzó con la aplicación de OMT. Los aspectos rigurosos y formales de Catalysis (como la influencia de Z y VDM) tenían precedentes en otros métodos de desarrollo orientado a objetos, como Fusion [Cm93] o Syntropy [Ck94]. Tuvo particular influencia el trabajo de Wills en su tesis doctoral [Ws92], dedicada a la aplicación de métodos formales en la orientación a objetos. La importancia que Catalysis otorga a las colaboraciones y contratos tiene precedentes, entre otros, en el trabajo de Trygve Reenskaug con OORAM [RWA96].

Catalysis es una metodología relativamente extensa, y no resulta fácil de resumir. Pero sus propios autores lo han intentado, y según ellos básicamente las características de Catalysis son:

- Las decisiones más importantes se pueden separar de las más detalladas. Lo que se hace, quién lo hace y cómo se hace son cuestiones separables.
- Los estados de los objetos se modelan con asociaciones y atributos.
- Las acciones se describen en términos de sus efectos sobre los objetos. Pueden definirse con postcondiciones (o diagramas de estado) e ilustrarse con instantáneas o *snapshots* (cierto tipo de diagramas).
- Las especificaciones abstractas pueden hacerse muy precisas, evitando ambigüedades.

- Las acciones y los objetos pueden generalizarse y refinarse (es decir, describirse a diversos niveles de detalle).
- El desarrollo se separa en una serie de capas, que tratan con análisis de negocio, especificación de requisitos, componentes, y diseño de objetos.
- Las plantillas (*templates*) son abstracción de modelos similares.
- Las colaboraciones (esquemas de interacción) son elementos de diseño "de primera clase".
- Los componentes y objetos se diseñan de manera similar, aunque con diferentes prioridades en la forma en que se eligen y se asignan las responsabilidades.
- Los componentes pueden diseñarse para encajar entre sí y también con el marco de referencia (*framework*). Los puntos de conexión se definen con especificaciones de acción.
- Una arquitectura de componentes define un *kit* estableciendo las convenciones de interoperabilidad, que se representan mediante conectores.

A partir de estas características, y siempre según los autores, se obtienen los siguientes beneficios:

- **Diseño de gran envergadura**. La separabilidad de las diferentes capas de decisión hace Catalysis especialmente adecuado para diseño en proyectos de gran importancia.
- **Diseño de alta integridad**. La precisión de las especificaciones de Catalysis las hace apropiadas para diseñar sistemas críticos y software empotrado.
- **Trazabilidad**. El refinamiento de Catalysis permite separar los modelos abstractos de las diversas implementaciones posibles. Los modelos abstractos son lo suficientemente precisos para ser puestos en relación con implementaciones específicas.
- **Reutilización de patrones y plena extensibilidad**. Los marcos de referencia de Catalysis se pueden usar para definir patrones de modelos específicos del dominio, protocolos de colaboración, y arquitecturas componente / conector.
- **Desarrollo basado en componentes**. Los componentes pueden ser especificados por una parte, desarrollados por otra, y utilizados por otra. Todas estas partes deben comprender las especificaciones. Catalysis extiende la clara especificación de componentes con la abstracción de conectores, simplificando el diseño de productos basados en componentes.
- **Un enfoque centrado en el comportamiento y en los datos**. Otros métodos, como OMT, han sido criticados por centrarse en el comportamiento. En Catalysis las dos vistas (comportamiento y datos) se apoyan entre sí con facilidad. Se describe el comportamiento de un componente en términos de atributos que se relacionan con los intereses de los clientes.
- **Soporte de herramientas**. Catalysis permite un alto nivel de soporte de herramientas mucho más allá de una simple base de datos de diagramas con

capacidad de generación de documentos. Las notaciones estándar y la clara relación entre artefactos también significa que se pueden usar herramientas populares de modelado UML en un proceso Catalysis respetando simples directrices de uso.

Una visión alternativa de los beneficios que aporta Catalysis se ofrece en [CA00]:

1. Precisión.
2. Facilidad de adaptación de la metodología sin ambigüedades ni pérdida de significado.
3. Vocabulario y notación para los componentes ya existente y documentada.
4. Refinamiento flexible pero preciso.
5. Un proceso para desarrollar, documentar e integrar componentes.
6. Un proceso que permite ir desde la captura de necesidades de negocio hasta una solución basada en componentes.
7. Un enfoque a la arquitectura del software elegante, práctico y que soporta bien el cambio de escala.
8. El reconocimiento de que los diseños y especificaciones son activos importantes.

Catalysis es, por tanto, una metodología de desarrollo, que cubre muchas de las áreas del análisis y diseño orientados a objetos tradicionales. Pero en Catalysis los componentes ocupan un lugar importante; examinemos ahora con más detalle las propuestas de Catalysis respecto a los componentes.

En Catalysis se distinguen diferentes tipos de componentes, o diferentes puntos de vista sobre los mismos:

**Def. VIII**: **Desarrollo basado en componentes** (*Component-Based Development*, CBD) es un enfoque del desarrollo de software en el cual todos los artefactos (desde código ejecutable a las especificaciones de interfaces, arquitecturas y modelos de negocio, y en tamaños que van desde aplicaciones y sistemas completos hasta pequeñas piezas) pueden construirse ensamblando, adaptando y conectando entre sí componentes previamente existentes en muy diversas configuraciones.

**Def. IX**: **Componente (general)**: Un paquete coherente de artefactos software que pueden desarrollarse y distribuirse de manera independiente como una unidad y que pueden componerse, sin cambios, con otros componentes para construir algo mayor.

**Def. X**: **Componente (en código)**: Un paquete coherente de implementación software que a) puede desarrollarse y distribuirse de manera independiente, b) tiene interfaces explícitas y bien especificadas para los servicios que proporciona, c) tiene interfaces explícitas y bien especificadas para los servicios que espera de otros, y d) puede componerse con otros componentes, quizás adaptando algunas de sus propiedades, sin modificar los componentes propiamente dichos.

Esto se complementa con la Def. VI (pág. 8), que es una definición informal de los componentes de implementación (aquí llamados "en código"). Un paquete de componente suele incluir lo siguiente:

- Una lista de interfaces proporcionadas.
- Una lista de interfaces requeridas.
- La especificación externa.
- El código ejecutable.
- El código de validación.
- El diseño.

Respecto al modelo de componente y de conexión que propone Catalysis, básicamente puede decirse que un componente expone interfaces; puntos del componente que representan los servicios ofrecidos y requeridos (llamados *sockets* y *plugs*, y genéricamente "puertos"). Un *plug* puede conectarse con un *socket* mediante un conector apropiado; en Catalysis, los conectores son elementos de considerable importancia. Se definen como:

**Def. XI**: Son **conectores** las conexiones entre puertos que permiten construir, mediante un conjunto de componentes, un producto software (o un componente mayor). Un conector impone restricciones específicas (ligadas al papel jugado) sobre los puertos que conecta, y se puede refinar dando lugar a protocolos de interacción concretos que implementan la unión.

En Catalysis, los conectores se definen como marcos de colaboración genéricos (un recurso previamente existente en la metodología). Gracias a esto se pueden modelar las complejas interacciones entre los componentes; los conectores ocultan la complejidad de estas colaboraciones, de modo que no es necesario definir cada colaboración desde cero en cada puerto de cada componente, sino que se recurre a una especie de *catálogo de conectores*.

Al presentar cierto conjunto de técnicas, los autores de Catalysis proponen ciertos patrones de actuación. Como orientación a las formas de actuación que se esperan al trabajar con componentes en Catalysis, cabe decir que en el capítulo sobre componentes y conectores de [SW99], los patrones propuestos son:

- Extracción de componentes de código genéricos.
- Gestión de los componentes.
- Construir modelos de marcos de referencia.
- Idoneidad de las conexiones (*plug conformance*).
- Uso de componentes heredados o de terceras partes.

## 2.9. Plataformas de componentes

Cuando se habla de "componentes software" entre profesionales del desarrollo se suele dar por hecho que se está haciendo referencia a lo que en esta tesis se denomina *plataformas de componentes*. Existen varias razones para esto:

- Estas plataformas de componentes se adhieren con gran fidelidad a la definición de componente software más extendida, mencionada en la introducción de esta tesis (Def. I y Def. IV, pág. 8).

- Además, estas plataformas son uno de los aspectos del desarrollo basado en componentes que *ya funciona*, que está ya disponible para su uso en proyectos reales.
- Las plataformas de componentes han recibido un grado notable de promoción y atención que ha llevado incluso a la existencia de varios enfoques competidores.

Szyperski llama a estas plataformas de componentes *estándares de "cableado"* [Sz97, p. 171]. La idea es que el papel de estas plataformas es, precisamente, facilitar la interoperabilidad entre distintos objetos o componentes.

Tómese la Def. I y sobre todo la Def. IV que la acompaña. Se aprecia que desde ese punto de vista se considera a los componentes como entidades esencialmente binarias, ya compiladas. La combinación de este tipo de elementos no siempre ha sido posible; por ejemplo, en los programas del sistema operativo MS-DOS no estaba previsto *fragmentar* el código ejecutable en unidades que pudiesen combinarse. La introducción de los llamados *overlays automáticos* por parte de la casa Borland en sus compiladores [TP86, OS98] fue un caso notable de fragmentación del código ejecutable en su segmento de mercado, pero su propósito no era ni mucho menos ofrecer unidades intercambiables de funcionalidad, sino que se trataba de un recurso técnico para evitar las estrechas limitaciones que imponían la escasez de memoria y el esquema de direccionamiento de los PCs originales.

Está claro, pues, que para poder trabajar con componentes binarios se necesita en primer lugar una infraestructura, que en el caso de los ordenadores personales no siempre estuvo disponible. Las **bibliotecas de enlace dinámico** de Windows (DLL, de *Dynamic Link Library*) [LR92] fueron un paso decisivo en esta dirección, abriendo el paso al uso de componentes en el mercado de los ordenadores personales. Pero estas DLL seguían planteando dificultades: el proceso de enlazado podía ser incapaz de combinar DLLs de diferentes compiladores debido a la llamada *decoración de nombres* (*name mangling*), sobre todo si se estaba trabajando con objetos, amén de otros problemas. De hecho, este tipo de dificultades fueron un factor importante de motivación para la aparición del modelo COM [Ro92, Hr95]; aparte de resolver este problema, el modelo definió procedimientos básicos de interoperabilidad.

Las tres principales corrientes de la industria por lo que se refiere a estos estándares de cableado son:

- COM (*Component Object Model*), de Microsoft.
- CORBA (*Common Object Request Broker Architecture*), de OMG.
- JavaBeans, de Sun Microsystems.

### 2.9.1. Precedentes: OSF DCE

Tradicionalmente, las formas de comunicación entre procesos que proporcionaban los sistemas operativos eran los ficheros, tuberías o *pipes*, *sockets*, semáforos, memoria compartida o alguna variante o combinación de estos. Yendo más allá de la comunicación entre procesos, sólo los *sockets* podían considerarse portables.

Una ventaja de estos esquemas es que –a excepción de la memoria compartida- podían fácilmente extenderse para funcionar a través de redes, lo que permitía comunicar a programas que residiesen en máquinas diferentes. No obstante, estos mecanismos están

basados en una transmisión puramente binaria, no estructurada, lo que podía conducir a errores y era, además, relativamente costoso de programar.

Para eliminar estos problemas, en 1984 se propuso el modelo de invocaciones a procedimientos remotos, o *Remote Procedure Calls* (RPC) [BN84]. El propósito de este modelo es que un programa realiza invocaciones a procedimientos que tienen el aspecto de llamadas normales, pero que en realidad provocan la llamada a un *stub*, un fragmento de código que se ocupa de "empaquetar" los parámetros de la llamada y enviarlos por red a su destinatario, en el que tiene lugar el proceso inverso en el *stub* receptor que a su vez invoca al procedimiento de destino. Tanto el llamador como el llamado utilizan convenciones de llamada propias y aparentemente están realizando y recibiendo llamadas locales, aunque en realidad esté teniendo lugar una comunicación entre ambos.

El *Distributed Computing Environment* (DCE) [DCE98] es un estándar de la *Open Software Foundation* (OSF) [OSF] que implementa RPC y permite la comunicación entre diferentes plataformas. Con el fin de automatizar la generación de código para los mencionados *stubs*, DCE desarrolló un lenguaje para describir los parámetros y tipos de los procedimientos involucrados, el llamado lenguaje de definición de interfaces (IDL, *Interface Definition Language*). Mediante este lenguaje se puede especificar el número, tipo y modo de paso (entrada o salida) de los parámetros y de los valores de retorno. Los tipos de datos involucrados tienen rangos y características claramente definidos, con el fin de que las llamadas sean coherentes entre diferentes plataformas.

Para identificar de manera inequívoca las entidades involucradas, DCE también introdujo el concepto de Identificadores Únicos a Nivel Universal (UUIDs, *Universally Unique IDentifiers*). Se trata de números de gran longitud, generados mediante un algoritmo que, a todos los efectos prácticos, garantiza su unicidad.

A pesar de representar un considerable avance, DCE no resolvía todos los problemas de interoperabilidad. Al fin y al cabo, es un esquema más bien procedimental en lugar de basado en objetos, y deja abiertas muchas cuestiones por lo que se refiere a interfaces, su identificación, la gestión de versiones, etc.

### 2.9.2. COM (Component Object Model)

El Modelo de Objetos Componentes [Ro97] es la base de todo el software basado en componentes de Microsoft. Asimismo, existen implementaciones de este modelo para otras plataformas (Apple, por ejemplo). El enfoque de COM es basado en objetos; los componentes COM reproducen el modelo de un objeto, aunque COM no soporta la herencia de implementación.

Hay que decir que en la plataforma de Microsoft se entremezclan diversas tecnologías, construidas sobre COM, y que además estas tecnologías han evolucionado con el tiempo, lo que ha creado una considerable confusión terminológica. En primer lugar, se ha producido la lógica evolución de nuevas versiones de la especificación, con la aparición de COM+ (una versión de COM que soporta el uso de transacciones sobre los componentes) o DCOM (soporte para objetos COM distribuidos en diferentes máquinas [Mi96]). Aparte de eso, las tecnologías construidas sobre COM han sufrido diversos cambios de nombre a medida que evolucionaban también, añadiendo confusión.

Originalmente, la tecnología que permitía crear documentos compuestos se denominaba OLE [Br96], que eran las siglas de *Object Linking and Embedding* (vinculación e incrustación de objetos). OLE 1.0 vio la luz en 1991, y no se basaba en COM, sino en el viejo protocolo de comunicación entre aplicaciones DDE (*Dynamic Data Exchange*) del sistema operativo Windows. DDE tenía fama de difícil de programar y poco fiable, y esta fama fue heredada por las versiones iniciales de OLE [Go98]. Conseguido el objetivo básico de los documentos compuestos, se enfocó OLE como una plataforma genérica para software basado en componentes y se rediseñó basándose en COM y no en DDE, lo que dio lugar a OLE 2.0 (1993). Esta nueva versión añadió funcionalidad para muchos otros propósitos. Fueron avances notables la automatización de aplicaciones (*OLE Automation*) o los controles OLE (OCX, *OLE Custom Controls*). Quedando las siglas OLE claramente anticuadas por su limitación, Microsoft afirmó entonces que OLE ya no eran siglas, sino simplemente el nombre que agrupaba a estas tecnologías. Puesto que los controles OLE tendían a tener un tamaño excesivo para su distribución por líneas lentas en Internet, en 1996 Microsoft emitió la especificación para los controles ActiveX, que podían ser mucho más pequeños e implementaban sólo las interfaces que necesitasen. ActiveX es el nombre que empezó a sustituir al antiguo de OLE en algunas áreas, y la confusión al respecto no ha desaparecido del todo.

A continuación figura una descripción general del modelo COM; pueden consultarse detalles en [DS99, Br93, Br96].

COM es un estándar binario que define cómo se crean y destruyen los objetos, y cómo interactúan entre sí. Al ser un estándar binario, permite que las aplicaciones que lo cumplen se comuniquen entre sí, independientemente del lenguaje de implementación elegido; sólo se necesita que dicho lenguaje soporte de alguna forma la creación de tablas de punteros. De hecho, un uso frecuente de COM es simplemente aprovechar las capacidades de comunicación inter-proceso que proporciona.

**Def. XII**: Una **interfaz COM** es una colección de métodos lógicamente relacionados que expresan cierta funcionalidad. Todas las interfaces COM derivan de IUnknown, y se identifican mediante un Identificador de Interfaz único a nivel global (IID).

**Def. XIII**: Una **clase COM** es una implementación de uno o más interfaces COM. Las clases se identifican mediante un Identificador de Clase único a nivel global (CLSID).

**Def. XIV**: Un **objeto COM** es una instancia de una clase COM.

Los CLSID e IID son, como ya se ha dicho, números de 128 bits que identifican las clases e interfaces en COM; son casos particulares de los UUID introducidos por OSF DCE.

Cuando un objeto COM (vale decir, componente COM)[3] necesita hacer uso de otro, solicita al sistema (concretamente a las extensiones COM del sistema operativo) que cree una instancia; para ello, deberá suministrar el CLSID de la clase de dicha instancia. El sistema carga entonces en memoria el módulo ejecutable que sea necesario (y que implementa la clase mencionada), y devuelve al objeto solicitante un puntero a la interfaz IUnknown del objeto recién creado.

[3] En realidad, un componente COM (entendido como fichero físico) puede implementar varios objetos COM, pero conceptualmente eso no tiene mayor importancia.

La interfaz IUnknown implementa funciones para la destrucción del objeto y también para la llamada *negociación de interfaces*; es la "puerta de entrada" para que el objeto solicitante pueda pedir al objeto creado una interfaz específica (identificada esta vez mediante su IID). En caso de que el objeto efectivamente implemente dicha interfaz (porque un objeto puede implementar varias interfaces), devolverá su puntero al solicitante, que podrá entonces invocar a los diferentes métodos o funciones de la interfaz.

Dado un CLSID, el sistema es capaz de determinar qué módulo ejecutable debe cargar en memoria porque los objetos COM deben estar **registrados** en una base de datos del sistema operativo (el llamado *registry* o registro del sistema); para un CLSID dado, en el *registry* figura -entre otros datos- la ruta completa del módulo ejecutable que le corresponde.

Este mecanismo es la base de la plataforma de componentes de Microsoft. Sobre él, Microsoft ha definido interfaces estándar para diversos propósitos, entre los que cabe citar:

- Controles ActiveX (objetos que actúan como elementos de interfaz de usuario añadidos a los del sistema operativo o bien como simples proveedores de funcionalidad)
- Creación de documentos compuestos (en los que intervienen elementos, tales como imágenes, textos, vídeos, etc. desarrollados por aplicaciones diferentes y desconocidas entre sí)
- Automatización de aplicaciones (gracias a la cual es posible "contactar" con una aplicación, tal como un procesador de textos o una hoja de cálculo, y utilizar su funcionalidad desde otros programas)

Los módulos ejecutables que implementan estas diferentes interfaces pueden así ser utilizados integrándolos con otros. Repasar todas estas técnicas con detalle queda fuera del propósito de esta disertación; basta dejar claro que todos estos casos de aplicación más complejos no son más que usos particulares (algunos muy complejos) del modelo subyacente, que es COM.

Las siguientes versiones (DCOM) permiten extender estas ideas a un entorno distribuido, en el que los objetos pueden residir en máquinas diferentes; la idea es similar, sólo que COM se ve ampliado para manejar la comunicación no ya entre procesos, sino entre máquinas diferentes. Asimismo, ofrece mecanismos que permiten al sistema encontrar e instanciar la clase requerida no en una sola máquina, sino entre varias.

COM ofrece mecanismos de reutilización, llamados continencia (*containment*) y agregación (*aggregation*). No obstante, no ofrece herencia de implementación; los dos mecanismos mencionados consisten (con algunas diferencias) en una yuxtaposición de objetos que colaboran para ofrecer la funcionalidad de cierta interfaz.

Otra particularidad de COM que merece la pena destacar es la existencia de interfaces salientes (*outgoing interfaces*); ciertos objetos COM proporcionan información, y por tanto permiten a otros objetos "escuchar" (*listener objects*) esta información. Para dar soporte a este protocolo, se han definido la interfaces `IConnectionPointContainer` e `IConnectionPoint`.

Para verificar la corrección al ensamblar componentes COM, este modelo se apoya en el IDL originalmente propuesto por DCE, lo que permite especificar las interfaces de los

componentes involucrados y detectar errores de tipos o número de parámetros en las invocaciones.

A nivel general, las ventajas de COM son, pues:

- La relativa independencia entre los componentes, ya que estos pueden ser desarrollados en lenguajes de programación diferentes siempre y cuando cumplan el estándar binario que propone COM.
- La facilidad de instalación y configuración, puesto que el uso de CLSID permite una referencia no ambigua a los componentes necesarios.
- La posibilidad de una comunicación relativamente sencilla entre procesos e incluso entre máquinas.
- La estandarización del proceso de integración de objetos, de modo que basta con hacer que se adapten al modelo para que puedan interactuar libremente.
- La existencia de multitud de interfaces y protocolos ya definidos para una amplia funcionalidad, que los nuevos componentes sólo tienen que implementar para poder integrarse de manera efectiva con aplicaciones ya existentes o aún por crear.

Como desventajas principales, cabe mencionar que la eficiencia de un programa (en términos de memoria y velocidad) se ve afectada por el uso de COM, y que la programación en COM/OLE también gana notablemente en complejidad.

### 2.9.3. CORBA (Common Object Request Broker Architecture)

CORBA fue definido por OMG (*Object Management Group*), una importante organización sin ánimo de lucro cuyo objetivo es definir todo tipo de estándares destinados a mejorar la interoperabilidad entre objetos. Su acción se ha dejado sentir en multitud de campos, desde las bases de datos a las plataformas de componentes, que es lo que aquí nos ocupa.

El objetivo de OMG al crear CORBA era precisamente abordar el problema de que objetos construidos en diferentes lenguajes, funcionando en diferentes sistemas operativos y/o máquinas, colaborasen. Como ya se ha mencionado, incluso la interacción de objetos implementados en el mismo lenguaje (C++) y funcionando en la misma máquina podía ser un problema si se habían utilizado compiladores distintos. La primera versión de CORBA, 1.1 (1991), trataba simplemente de solucionar estos problemas de interacción básica. En julio de 1995 se publicó CORBA 2.0, que se actualizó en julio de 1996 [OMG97] y especificó cómo podían interactuar las implementaciones CORBA de diferentes fabricantes. CORBA 2.5 (2001) incluyó especificaciones sobre tolerancia a fallos, tiempo real, minimumCORBA (una versión especial para sistemas empotrados que no necesitaban aspectos dinámicos) y otras mejoras. CORBA 3 está en preparación y entre otros avances incluye una mejor integración con Java e Internet (especialmente con Enterprise JavaBeans), un control más estrecho de la calidad de servicio, y CORBAcomponents / CORBAscripting, que proporcionan un entorno contenedor para los componentes y un formato de distribución que permite la existencia de un mercado de componentes. Los detalles sobre las especificaciones CORBA pueden consultarse en [OMG].

En palabras del OMG, CORBA es la respuesta de dicho organismo a las necesidades de interoperabilidad entre la gran variedad de productos hardware y software que hoy

proliferan. CORBA permite a las aplicaciones comunicarse entre sí, con independencia de cuál sea su ubicación o quién las haya diseñado.

El ORB (*Object Request Broker*) es el *middleware* que establece las relaciones cliente / servidor entre objetos. Mediante un ORB, un cliente puede invocar de manera transparente un método de un objeto servidor, que puede estar en la misma máquina o al otro lado de una red. El ORB intercepta esta llamada y se ocupa de encontrar un objeto que pueda implementarla, pasarle los parámetros, invocar su método y devolver los resultados. El cliente no necesita saber dónde se ubica el objeto servidor, su lenguaje de programación, su sistema operativo o cualquier otro aspecto que no sea parte de la interfaz de dicho objeto. De este modo, el ORB proporciona interoperabilidad entre aplicaciones, en diferentes máquinas y en entornos distribuidos heterogéneos.

En las aplicaciones cliente / servidor tradicionales, los desarrolladores diseñan el protocolo que utilizan, o bien eligen un estándar. La definición del protocolo depende del lenguaje de implementación, el transporte de red y muchos otros factores. Los ORB simplifican este proceso; el protocolo viene definido por las interfaces de la aplicación gracias a que CORBA también utiliza IDL, que es una especificación independiente del lenguaje. Del mismo modo, el programador gana la flexibilidad de elegir el sistema operativo, entorno de ejecución y lenguaje de programación más adecuados para cada componente. Los componentes ya existentes, además, pueden integrarse, aun cuando no sean orientados a objetos; basta modelarlos basándose en interfaces y luego escribir código de envoltorio.

En CORBA todo está cuidadosamente estandarizado, pero no es un estándar binario; esto permitió que muchos fabricantes diferentes aportaran implementaciones, pero como contrapartida los objetos de CORBA no pueden comunicarse de forma plenamente eficiente a un nivel binario, sino que deben hacerlo mediante protocolos de más alto nivel y menos eficientes.

Un protocolo de especial importancia es el Protocolo Inter-ORB de Internet (*Internet Inter-ORB Protocol*, IIOP), introducido en CORBA 2.0. Este es el protocolo que deben cumplir todos los ORB que afirmen guardar interoperabilidad con otros sistemas.

Los objetos *servidores* se registran en un ORB, y desde ese momento el ORB "sabe" cómo activar dicho servidor cuando sea necesario. Los clientes que no ofrezcan ningún objeto como servidor no necesitan registrarse.

Puede verse que CORBA y COM (DCOM, para ser más exactos) responden, en muchos casos, a necesidades similares, auque la forma concreta de hacerlo varíe. Al igual que en el caso de COM, lo descrito hasta aquí es un modelo de cableado básico; se limita a definir (que no es poco) cómo un objeto puede crear otro e invocar sus métodos. Pero a un nivel más alto de abstracción, existía un vacío. Microsoft definió toda una arquitectura (englobada bajo los nombres de OLE y/o ActiveX) encaminada a rellenar este hueco y proporcionar servicios más específicos; del mismo modo, el OMG definió OMA, la Arquitectura de Gestión de Objetos (*Object Management Architecture*) [OMG97b]. Esta especificación añade definiciones sobre servicios (CORBAservices), utilidades (CORBAfacilities) y aplicaciones. CORBAservices incluye especificaciones sobre eventos, ciclo de vida, persistencia, transacciones, seguridad, licencias, etc. CORBAfacilities se centra en elementos de más alto nivel, como documentos compuestos, gestión del escritorio, interfaces de usuario, ayuda, etc.; y los objetos de aplicación se corresponden con estándares emitidos por organizaciones de mercados verticales.

Nuevamente, no es este el ámbito apropiado para abordar en detalle la complejidad de CORBA; basta una idea general del papel de este modelo en lo que pueda tener que ver con el tema de esta tesis. En el plano general que aquí se busca, las principales ventajas de CORBA son [Vi97]:

- **Heterogeneidad**. El uso de IDL permite definir las interfaces de manera independiente del lenguaje, y los objetos CORBA pueden distribuirse en plataformas muy diversas.
- **Modelo de objetos**. La interacción entre objetos está definida de modo que es indiferente el protocolo de red subyacente.
- **Integración de lo ya existente**. La especificación de CORBA no hace referencia a la implementación; es posible integrar protocolos y aplicaciones existentes, como DCE o COM.
- **Enfoque orientado a objetos**. CORBA (empezando por IDL) está diseñado según el paradigma de la orientación a objetos, lo que hace que las aplicaciones implementadas sobre CORBA se beneficien también de este enfoque.

### 2.9.4. JavaBeans

Sun Microsystems sacó a la luz una versión alfa del lenguaje Java [AG97] en 1995. Poco después, comenzó a extenderse su uso y hoy en día es uno de los lenguajes de programación más utilizados. Sus características esenciales pueden resumirse así:

- Es un lenguaje orientado a objetos "casi puro".
- La sintaxis es parecida a la de C++.
- La gestión de memoria se realiza por medio de un recolector de basura (*garbage collector*), lo que elimina muchos de los problemas tradicionales de C++ que exigía técnicas de programación rigurosas.
- Se trata de un lenguaje multiplataforma. Los programas Java se compilan a un código intermedio estándar (el llamado *bytecode*) que luego se interpreta en la máquina de destino.
- El programa intérprete de este código es la llamada máquina virtual de Java (*Java Virtual Machine*, JVM).
- Para evitar la ineficiencia que el proceso de interpretación puede producir, existen esquemas de compilación del *bytecode* a código nativo del procesador (como la compilación *just in time*, "en el momento") que luego se ejecuta normalmente.
- El lenguaje ofrece soporte relativamente avanzado al programador: bibliotecas, hilos y sincronización, comunicaciones, etc.
- Java cuenta con algunas construcciones de especial relevancia en Internet, como los Applets, y en el mundo de los componentes software, como los JavaBeans.

Las principales razones del éxito de Java y de su gran difusión en el desarrollo relacionado con Internet probablemente sean, por un lado, la existencia del *bytecode* y de la máquina virtual (lo que permite desarrollar programas sin preocuparse del hardware o el sistema

operativo, ya que podrán ejecutarse en cualquier máquina que tenga instalada la máquina virtual de Java), y por otro lado los llamados applets.

Un applet es una pequeña aplicación, implementada como un objeto, que se ejecuta dentro de una página web (y en el ordenador cliente). En dicha página puede incluirse código –en forma de *scripts*- que invoca los métodos y propiedades del applet; este dispone de cierta zona de la página (el equivalente a una ventana dentro de la misma) en la que puede mostrar la información (frecuentemente gráfica). Evidentemente, la potencia que tiene un lenguaje de programación como Java llega mucho más allá que el mero HTML, por lo que un applet permite ofrecer funcionalidad relativamente avanzada.

El applet se descarga de manera automática junto con la página web que lo contiene (igual que cualquier imagen estática), por lo que el usuario no necesita preocuparse de tareas de instalación para conseguir la funcionalidad que el applet le ofrece. Además, la seguridad es un aspecto muy cuidado: para evitar que código malicioso pueda afectar al ordenador cliente, la JVM cuenta con un sistema de supervisión que impide al applet ejecutar cualquier código potencialmente dañino (por ejemplo, crear o borrar ficheros en el disco duro del equipo cliente).

Aunque los applets encajan, hasta cierto punto, con la noción de componente software dada por las definiciones precedentes (pág. 8), los verdaderos "componentes" de Java son los JavaBeans. Los applets no dejan de ser mini-aplicaciones; son reusables, la página web puede interactuar con ellos, pueden provenir de terceros, pero los applets de una página no pueden interactuar entre sí. Por tanto, los applets son entes relativamente aislados.

En octubre de 1996 aparecieron los JavaBeans [Su97], con el propósito de ser los "componentes de Java". El término *bean* se refiere tanto al *componente* (las clases y recursos que necesita) como a la *instancia* del componente. En principio, un bean puede considerarse frecuentemente como un control, un elemento con una interfaz de usuario visible, aunque es posible que tal interfaz no exista y el bean ofrezca su funcionalidad sin ningún comportamiento visual. Se supone que los beans pueden también integrarse en un entorno contenedor externo a Java; por ejemplo, existen adaptadores que permiten funcionar a un bean como control ActiveX.

Los beans se manejan en dos etapas diferentes. En "tiempo de diseño", quien crea la aplicación crea instancias de los beans necesarios en algún tipo de herramienta contenedora, y conecta entre sí a estos componentes; después, el programa así diseñado se utiliza en "tiempo de ejecución". El propio bean puede (suele) comportarse de manera distinta en ambos casos.

Los principales aspectos del modelo de *beans* son [Sz97]:

- **Eventos**. Los *beans* pueden anunciarse como generadores o receptores de ciertos tipos de eventos. La herramienta de ensamblaje puede utilizar esta información para conectarlos con otros beans. Los objetos interesados en un evento se añaden a una "lista de oyentes", y cada vez que el bean genere ese evento todos los oyentes recibirán la notificación correspondiente.
- **Propiedades**. Cada objeto expone una serie de propiedades. El cambio de valor de una propiedad puede ser denegado por el bean; también puede provocar el disparo de eventos.

- **Introspección**. Las herramientas de ensamblaje pueden interrogar al bean para saber qué propiedades, eventos y métodos ofrece.
- **Personalización**. Mediante la herramienta de ensamblaje, se puede personalizar el aspecto y/o comportamiento de un bean, alterando los valores de sus propiedades.
- **Persistencia**. La estructura de instancias de beans que se han creado y conectado en la herramienta de ensamblaje puede guardarse en disco; en tiempo de ejecución, esta información se cargará para crear la aplicación.

Los beans pueden estar formados por varias clases y recursos; en el caso de los applets, estos elementos se descargan uno por uno del servidor, pero los beans pueden ser empaquetados de modo que todo lo necesario esté en un archivo JAR (Java Archive). Un JAR no es más que un archivo comprimido, que puede contener además un archivo (el *manifest file*) que proporciona información sobre lo que contiene el JAR. Un archivo JAR puede contener los recursos siguientes:

- Ficheros de clase.
- Objetos serializados (que pueden servir como inicialización de los objetos).
- Archivos de ayuda en HTML.
- Información de localización (para entornos internacionales).
- Imágenes de icono (en formato GIF).
- Otros ficheros que el bean necesite.

Merece la pena destacar también que un fichero JAR puede contener varios beans con sus recursos asociados.

Haciendo hincapié en aspectos de interacción entre componentes, hay que hacer notar también que el lenguaje Java ofrece un mecanismo de objetos distribuidos, basado por una parte en la serialización de objetos (que permite transmitirlos con facilidad, con lo que los parámetros de las llamadas remotas pueden ser en sí mismos objetos de relativa complejidad) y un servicio de invocación remota de métodos, el RMI (*Remote Method Invocation*). Para hacer referencia a un objeto remoto se utiliza siempre una interfaz –no se guarda información sobre clases ni superclases- y las interfaces accesibles de este modo derivan de una interfaz base, `java.rmi.Remote`. Como cualquier llamada a un método remoto puede fallar por problemas de red, todos los métodos remotos pueden lanzar la excepción `java.rmi.RemoteException`.

Otra característica especialmente importante de RMI es que el mecanismo de recolección de basura funciona de forma distribuida, basándose en el trabajo de Birrel y otros [BE93]. El mecanismo de recolección de basura distribuido es quizás la mayor diferencia entre el RMI de Java y las demás plataformas de componentes.

Aunque se ha mencionado repetidamente que la independencia del sistema operativo / hardware es una característica básica de Java, este lenguaje permite integrar elementos Java con otros específicos de la plataforma. Para ello, la interfaz nativa de Java (JNI – *Java Native Interface*) especifica, para cada plataforma, las convenciones de llamada nativas, lo que permite al código Java invocar código exterior a la máquina virtual (y viceversa, el código nativo puede invocar a código Java).

## 2.10. Diagramas-resumen de las tendencias analizadas

A continuación se muestran algunas ilustraciones que ofrecen un breve resumen de las distintas tendencias analizadas y su interrelación, incluyendo los grupos de investigación y autores más notables, de manera informal y no exhaustiva. (Nótese que el grupo CSSRG-CMU aparece en dos ilustraciones). En la Ilustración 11 se ofrece un cuadro-resumen que compara las técnicas evaluadas (👍 - adecuado, 👎 - inadecuado, ✋ - discutible). Téngase presente que esta valoración es una simplificación y además se realiza a la luz de nuestros objetivos particulares (no es una valoración global).

| | Verificación | Estaticidad | Automatismo | Facilidad de aprendizaje | Facilidad de implementación | Flexibilidad, aplicabilidad | Soporte componentes |
|---|---|---|---|---|---|---|---|
| Métodos formales | ✋ | 👍 | ✋ | 👎 | 👎 | 👎 | 👎 |
| Análisis estático / interp. Abstracta | 👍 | 👍 | 👍 | 👎 | 👎 | 👎 | 👎 |
| Especificación semántica | ✋ | 👍 | 👍 | 👎 | 👍 | ✋ | 👍 |
| Álgebras de procesos | ✋ | ✋ | ✋ | 👎 | ✋ | 👎 | 👎 |
| Lenguajes de composición | ✋ | ✋ | ✋ | ✋ | ✋ | 👎 | 👎 |
| Contratos (Meyer) | ✋ | 👎 | 👍 | 👍 | 👍 | 👎 | 👍 |
| Contratos bilaterales | 👎 | 👎 | 👎 | 👍 | ✋ | 👍 | 👍 |
| Contratos de reutilización | ✋ | 👍 | 👍 | 👍 | 👍 | 👎 | 👍 |
| Lenguaje Contract | ✋ | 👎 | 👍 | 👍 | ✋ | 👎 | 👍 |
| Estilos arquitectónicos | ✋ | 👍 | ✋ | 👍 | ✋ | 👎 | ✋ |
| ADLs | ✋ | ✋ | 👍 | ✋ | ✋ | 👎 | ✋ |
| Metodologías AOO/DOO | 👎 | 👍 | 👎 | ✋ | ✋ | 👍 | 👍 |
| Plataformas de componentes | 👎 | 👎 | 👍 | 👎 | ✋ | 👎 | 👍 |

*Ilustración 10. Cuadro-resumen de las técnicas evaluadas*

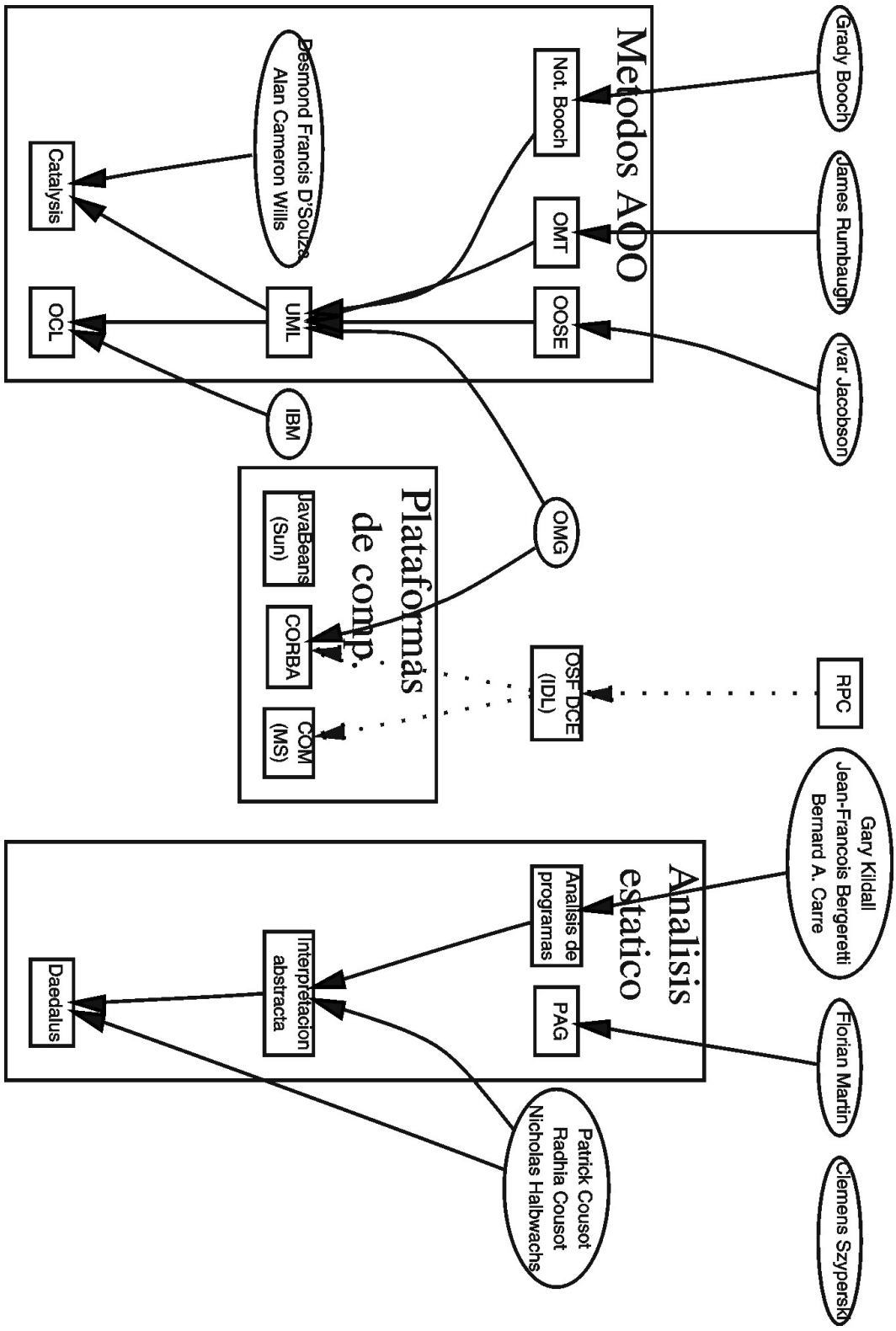



*Ilustración 11. Revisión de trabajos relacionados y antecedentes (I de III)*

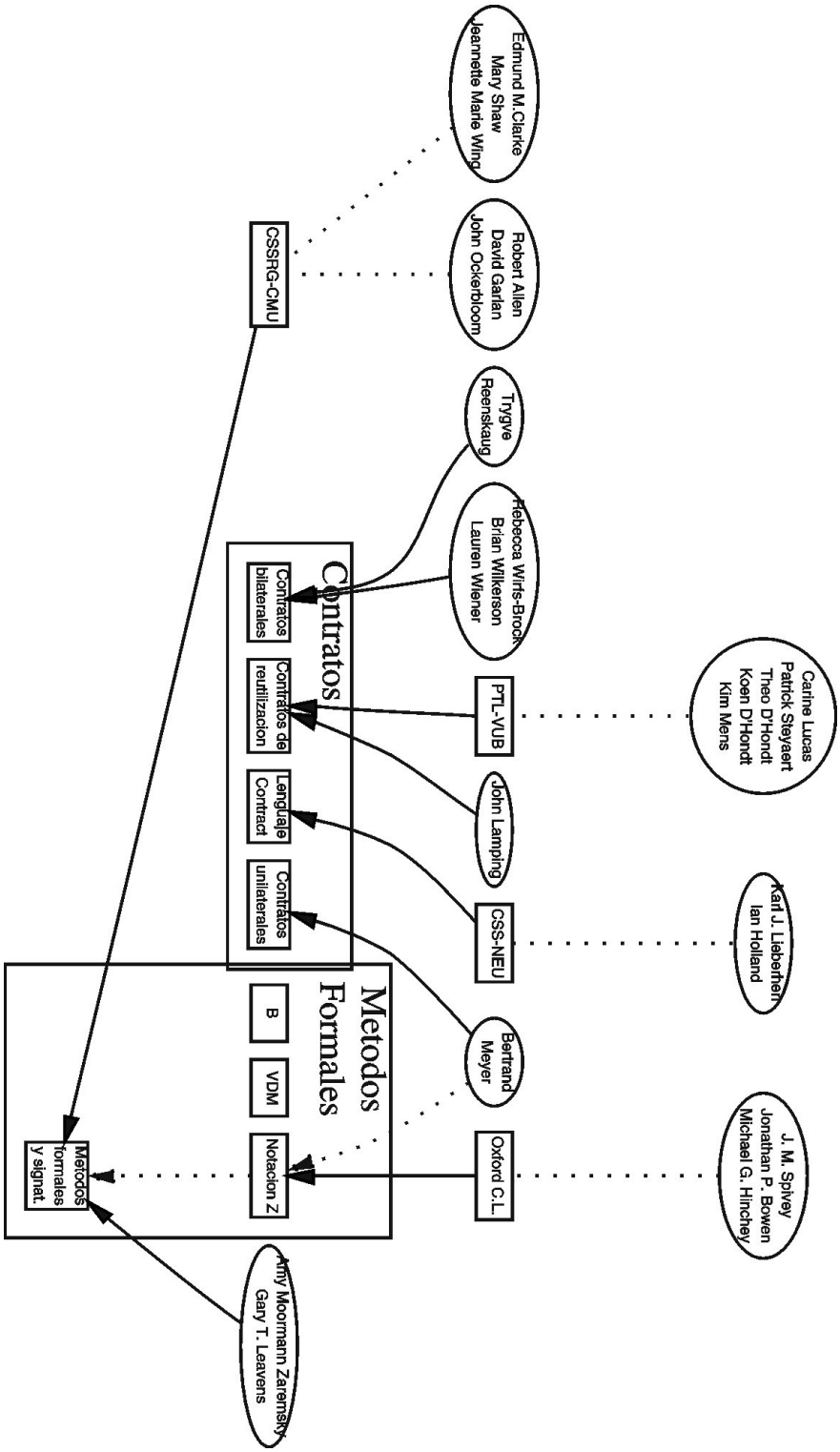



*Ilustración 12. Revisión de trabajos relacionados y antecedentes (II de III)*

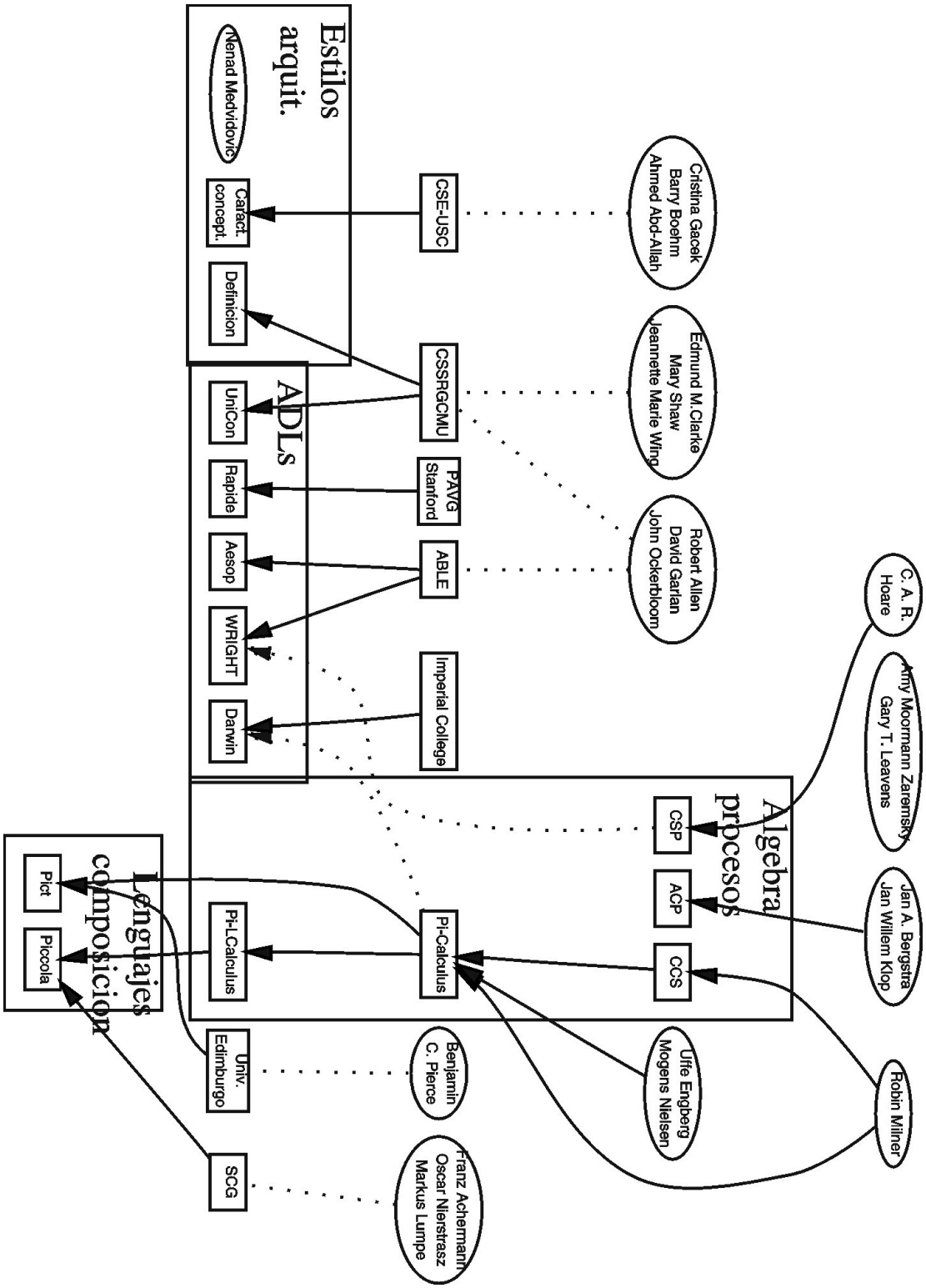



*Ilustración 13. Revisión de trabajos relacionados y antecedentes (III de III)*

# 3. Un sistema de verificación de componentes software

## 3.1. Descripción del problema

Como ya se ha dicho en la introducción de esta disertación, la construcción de software constituye un área fundamental del desarrollo técnico. En primer lugar, por estar el software presente en prácticamente todos los campos de la actividad industrial, económica o científica, y depender de él desde la calidad de servicio a los clientes hasta, en ocasiones, vidas humanas. Pero en segundo lugar, porque la construcción de software fiable y de alta calidad ha resultado ser un problema al que la ingeniería aún no ha dado una solución del todo satisfactoria.

El tema de la calidad del software es demasiado amplio para un análisis detallado que no es el objetivo de este documento. Pero dejando aparte muchos de los posibles enfoques relativos a la calidad y centrando la atención sólo en los defectos, en los meros errores de funcionamiento del producto final, ya se tiene delante un campo de actuación enormemente abierto y complejo.

Cabe distinguir diversos tipos de errores. Es conocido que los errores más problemáticos son los producidos en fases iniciales del desarrollo [Pr92, Mc96], que ven aumentada su gravedad con el paso del tiempo, mientras permanecen ocultos, y se manifiestan finalmente en un momento en el que su corrección resulta mucho más costosa. Se pueden señalar errores en la recogida de requisitos, errores de gestión y estimación del proyecto, errores de análisis, de diseño... y, por supuesto, los errores introducidos por modificaciones de requisitos funcionales *durante* el desarrollo del proyecto.

Estrechando aún más el campo, se pueden dejar a un lado los problemas de la comunicación humana o la gestión de proyectos y centrar la atención exclusivamente en los errores de construcción del software, y que podríamos denominar **defectos de fabricación**, es decir, los comportamientos no deseados por los diseñadores o constructores del sistema (que no provienen, por tanto, de una comprensión deficiente de los requisitos funcionales o de diseño). Es este un problema que afecta muy negativamente a la calidad del software, puesto que los errores de este tipo son, en muchos casos, extremadamente difíciles de detectar (pueden manifestarse sólo en situaciones muy concretas, que los casos de prueba no saquen a relucir [Rb00, pág. 54, citando a McConnell]), y sin embargo pueden tener graves consecuencias en el rendimiento y fiabilidad del sistema.

Como es sabido, el software es un artefacto de complejidad extraordinaria [Bo94]. Además, los sistemas basados en computadora son sistemas discretos cuya respuesta a situaciones anómalas puede resultar catastróficamente errática. Recogiendo terminología extraída de la teoría del caos, para variaciones pequeñas de las variables de entrada pueden producirse variaciones enormes en los resultados finales[Li96].

Por tanto, una pequeña desviación o error en un sistema de software puede tener efectos amplificados en muchos órdenes de magnitud, que en muchos casos producen la caída total de dicho sistema o interrumpen su actividad normal. Pero para empeorar las cosas, es altamente improbable que un sistema basado en computadora no esté afectado por errores, en muchos casos desconocidos hasta el momento en que se manifiestan afectando negativamente al sistema.

Algunas de las técnicas que habitualmente se aplican en la industria para afrontar el problema de la fiabilidad del software resultan, en cierto sentido, "paliativas". La prueba exhaustiva del software es, en general, técnicamente imposible o, en el mejor de los casos, económicamente inviable. La calidad en cuanto al número de defectos se gestiona, muy frecuentemente, mediante una fase de pruebas que cuenta con técnicas de "cobertura" o que trata de desafiar la capacidad del software para manejar tanto situaciones previstas como situaciones límite; también mediante técnicas de medición y extrapolación, de forma que el ingeniero puede verificar cuál es la tendencia del ritmo de aparición de defectos para tener una idea de cuándo un componente se está acercando a un punto de estabilidad y fiabilidad *suficientes*. Pero todas estas medidas en ningún caso garantizan la inexistencia de errores ocultos. Se dice popularmente que la prueba puede demostrar la presencia de defectos, nunca su ausencia.

No parece que el estado de la tecnología permita prever que se acerque una situación ideal en la que cualquier sistema software pueda "certificarse" como libre de defectos, al menos en un futuro cercano (entre otras cosas, debido a restricciones de tipo económico y práctico). Pero sería deseable mejorar progresivamente las técnicas dirigidas a aumentar la fiabilidad del software en un nivel lo más básico posible. Esto es así porque, aunque los recursos que se pueden dedicar al desarrollo son limitados, hay muchas áreas en las que actuar para poner en pie un sistema basado en computadora. La detección y corrección de defectos de programación y construcción se lleva una parte considerable de estos recursos, que darían un fruto mucho mayor si se dedicasen a la detección de errores de diseño, de captura de requisitos, de gestión del proyecto.

Hoy por hoy, parece una quimera la atractiva idea del *software correcto por construcción*. No obstante, existen algunas técnicas que pueden eliminar parte de los defectos de construcción hasta cierto grado. Un problema con el que frecuentemente se enfrentan estas técnicas es su coste de aplicación. Por poner un ejemplo, existen varios **métodos formales**, sobre los cuales se discute en el capítulo 3.2.1; allí se explica en qué medida la aplicación generalizada de estos métodos se tropieza con importantes obstáculos de diversa índole.

Otra tendencia que se suele apuntar como solución a muchos problemas de desarrollo y fiabilidad es el software **basado en componentes**. Se argumenta que si el software se construye a través de la composición de bloques "estancos", bien probados y que resuelven problemas típicos, aumentaría la fiabilidad y sobre todo se abarataría el coste de desarrollo, al ahorrarse gran parte del tiempo que requiere solucionar un problema que alguien ha solucionado antes. Es un tema de relativa actualidad, y por eso no resulta difícil encontrar un gran "ruido" provocado en parte por intereses comerciales. Es frecuente que detrás del

ruido lo que haya sea simplemente una solución técnica para comunicar entre sí elementos de software y ensamblarlos en cierta forma, pero sin ningún tipo de formalización o técnica que permita confiar en que los componentes son de mejor calidad que el software desarrollado desde cero, o que son siquiera apropiados para el uso que se les va a dar. Es más, los estudios serios sobre el tema revelan que en la combinación de componentes es donde se producen muchos defectos inesperados, porque aun disponiendo de componentes fiables y correctos, su combinación puede resultar incorrecta, desde el nivel elemental de ensamblaje hasta el más alto de combinación de estilos arquitectónicos [GA97].

El ideal que se persigue en este trabajo es aportar ideas que permitan avanzar algunos pasos en ese camino. La motivación del mismo reside en la eliminación de algunos de los defectos de *construcción* más frecuentes. Por poner un ejemplo, la tecnología de compiladores ha borrado prácticamente del mapa algunos de los defectos que probablemente se cometerían si el software se construyese mediante lenguajes de muy bajo nivel; piénsese en los problemas de corrupción de la pila del sistema. Desde el momento en que se liberó al programador de la tarea de recordar qué argumentos se ponían y quitaban en la pila, y se sustituyó esta forma de trabajar por el uso de rutinas con una signatura de llamada verificada en tiempo de compilación, la frecuencia de este tipo de errores disminuyó drásticamente. En un entorno de programación típico de lenguajes imperativos, por ejemplo, es difícil que se dé este hecho, a no ser en funciones con número de argumentos variable, en enlace dinámico en ejecución, o mediante la mezcla de bibliotecas con declaraciones de la convención de llamada incongruentes entre el llamador y el llamado. Obviamente, también puede corromperse la pila del sistema por otros medios, como sobreescribiendo accidentalmente la memoria; pero en cualquier caso, el mecanismo de llamada en sí mismo suele estar a salvo de todos estos problemas.

Por tanto, estos defectos de construcción han sido casi erradicados, y el trabajo de programación no se malgasta en la búsqueda de incoherencias de este tipo, que por lo general se detectan de forma automática en tiempo de compilación gracias a la estructura de los lenguajes de programación y los compiladores que les dan soporte. En este sentido, el mecanismo de llamada entre módulos es -en lenguajes con comprobación estricta de tipos y enlace temprano- *correcto por construcción*.

A pesar de los problemas señalados, conviene aclarar que creemos que las ideas ya mencionadas -el uso de métodos formales y el uso de componentes- son sin duda ideas valiosas de cara a conseguir software "correcto por construcción". El uso de componentes parece una vía razonable para conseguir dominar hasta cierto punto la complejidad del software gracias a la división y el encapsulamiento, y el éxito de las técnicas de orientación a objetos así lo hace pensar. Además, cabe esperar que un uso formalizado de los elementos utilizados para construir software permitirá una verificación más sistemática y completa de dichos elementos, en etapas tempranas y de forma estática, independiente así de las condiciones de ejecución y sirviendo de valioso complemento a las técnicas de prueba convencionales.

Siendo su disciplina muy joven en comparación con otros campos de la ingeniería, los ingenieros del software vuelven la vista con frecuencia hacia sus colegas de otros ámbitos, con el fin de reconocer en sus métodos de trabajo la huella de problemas que estos colectivos han solucionado tiempo atrás, y que en el terreno del software han hecho su aparición en épocas recientes. Por poner un ejemplo, después de años de construir el software de forma puramente artesanal y tras encontrarse con la creciente complejidad de

los sistemas basados en computadora, se vio la necesidad de afrontar su construcción a un nivel más alto, realizando un importante esfuerzo de análisis y diseño. Los ingenieros del software necesitaban *planos*. Aunque estas prácticas tardaron un tiempo en arraigar, y por desgracia aún hoy en día se desarrolla software de forma improvisada, las actividades de análisis y diseño parecen haber alcanzado un cierto grado de madurez, aunque aún queda mucho por mejorar en lo referente al *cómo* se realizan dichas actividades y a las técnicas disponibles para ello, siendo un campo de investigación en constante efervescencia. Al menos, lo cierto es que hoy se cuenta con diversos estándares y notaciones gráficas, que sientan las bases para un lenguaje común entre profesionales y son además una herramienta de valor incalculable para desarrollar software de calidad.

Del mismo modo, cabe afirmar que los ingenieros del software echamos de menos algunos recursos de los que disponen ingenieros de otras disciplinas. Por ejemplo, en la construcción de sistemas electrónicos los componentes involucrados tienen un comportamiento definido por ciertos modelos físicos bastante precisos. Estos modelos físicos, frecuentemente, se pueden caracterizar de manera cuantitativa a partir de ciertos parámetros de funcionamiento. Cuando uno de estos componentes se ubica en el diseño de un circuito, el ingeniero dispone de herramientas de simulación y análisis que le permiten realizar verificaciones automatizadas de su diseño. Esto le permite obtener ciertas conclusiones sobre el comportamiento del sistema antes de llevarlo a su implementación.

Por supuesto, el modelado de un componente físico no está sujeto a los mismos condicionantes que el de un componente software. Los componentes físicos son, frecuentemente, sistemas continuos y no discretos, y responden a leyes físicas conocidas. Los componentes software son sistemas discretos, lo que los hace más inestables y difíciles de modelar, y además su comportamiento responde a un propósito muy variable y difícil de caracterizar, y más aún cuantitativamente.

Pero dejando aparte esas apreciaciones, quizás a pesar de todo estos sistemas electrónicos presenten algunas propiedades que se puedan reproducir en cierta forma en los sistemas software, obteniéndose así algunas de las ventajas que hacen fiables a aquellos. Por ejemplo, los sistemas electrónicos citados se construyen casi siempre mediante componentes, y se construyen además con vocación de componentes; es decir, los elementos integrantes de un sistema electrónico tienen interfaces definidas con toda claridad, y especificaciones relativas a dichas interfaces, especificaciones cuyo cumplimiento se puede verificar. (Nótese que muchos de los elementos software que alegremente se califican como *componentes* en el plano comercial están muy lejos de encajar con tal descripción).

¿Cómo conseguir ventajas similares a la hora de construir sistemas software? Ese es el objetivo de este estudio y estas son las ideas que se desarrollarán a lo largo del mismo. Algunos puntos a tener en cuenta son:

- Actualmente, se acepta que un componente es un elemento de software con una interfaz, pero esa interfaz frecuentemente queda definida con una simple enumeración de signaturas -operaciones, propiedades, parámetros y tipos de los mismos- y una documentación descriptiva informal. Pero en esta disertación se sostiene que la signatura ocupa un nivel elemental, y que la interfaz de un componente debe llegar mucho más allá e incluir consideraciones funcionales cuyo cumplimiento sea susceptible de ser verificado en el momento del ensamblaje (en "tiempo de compilación").

- Esto requiere un planteamiento mucho más formal y rico que el que hoy en día se adopta para definir un componente, como es el caso del IDL (véase capítulo 2.9). No obstante, esa formalización debe realizarse de manera muy cuidadosa, puesto que un objetivo fundamental de este trabajo es proporcionar técnicas que sean aplicables *en el mundo real* y teniendo en cuenta las condiciones que establece el proceso de desarrollo de software. Se mejorará el grado de formalización, pero este no será nunca tan alto como para que estas técnicas puedan no ser operativas, problema al que se enfrentan muchos de los enfoques formales tradicionales (véase el capítulo 3.2.1 para una discusión completa sobre el particular). De hecho, en esta disertación se evita el término *formalización* al hablar de la solución propuesta, y se reserva para dichos métodos formales ampliamente conocidos.

- Los componentes deberían, pues, incluir una descripción explícita de los requisitos que plantean para sus entradas (ya sean estas en forma de llamadas, valores de los parámetros, etc.) y de las garantías que ofrecen sobre sus salidas; esto no deja de ser una forma de modelar su comportamiento, puesto que su interior se considera una "caja negra" y sólo se necesita información referente a sus conexiones con el mundo exterior.

- El tipo de restricciones que se necesita expresar puede ser muy variado; por nombrar sólo algunos ejemplos, podría aludir a versiones de los componentes, nombres de fabricantes, rangos de valores admisibles, datos de eficiencia o consumo de memoria, patrones de diseño, características de tipo arquitectónico o estructural, requisitos de inicialización o patrones de llamada, y en general cualquier tipo de exigencia o garantía que un componente pueda ofrecer o requerir sobre sus puntos de conexión. Claramente, esto va mucho más allá de las interfaces tradicionales, y exige una notación altamente flexible y abierta, que no plantee dificultades notables para su uso.

- El propósito principal de la especificación a la que nos referimos no es servir de descripción de diseño para la posterior construcción del componente (cosa que, por supuesto, no se descarta, y se entiende como un beneficio lateral). El fin primario de estas especificaciones para un componente A es que sea posible ponerlas en relación con las especificaciones de otros componentes a los cuales se conecta. De este modo, será posible comprobar si el componente B, conectado a A, satisface los requisitos planteados por A; análogamente, también se podrá comprobar si el componente B ve satisfechos sus propios requisitos a partir de las garantías que ofrece A.

- Esta comprobación se realiza de manera estática, es decir, sin necesidad de poner el sistema en ejecución. Las expresiones que describen el comportamiento del componente deben poder contrastarse con las expresiones de los componentes vecinos por sí solas, sin intervención del código ejecutable del componente. Esto permite detectar las incoherencias entre las especificaciones de manera inmediata, cuando se establece la interconexión de los componentes, sin necesidad de abordar las fases de pruebas. Así se evitará el riesgo de que los defectos que se puedan detectar de esta forma analítica lleguen a la fase de pruebas (o la superen).

- El mecanismo de comprobación no puede limitarse a contrastar las restricciones de un componente con las del componente vecino. Estas restricciones pueden dar lugar a dependencias transitivas, de manera que el comportamiento de un componente vecino se vea influido por el comportamiento de sus propios vecinos, y así sucesivamente. Por tanto, el modelo de verificación debe ser capaz de poner en relación las especificaciones

de los componentes incluso de manera indirecta o transitiva. En [SW99, pág. 299] puede verse un ejemplo de esta necesidad. Ante la posibilidad de que restricciones importantes se deriven de la combinación de otras ("la válvula de refrigeración se abre cuando la temperatura es >700", "cuando se abre la válvula de refrigeración suena una alarma") los autores proponen incluir explícitamente reglas redundantes ("cuando la temperatura es >700 suena una alarma") para que tal conocimiento no pase desapercibido. Parece importante que un sistema de verificación automatizado sea capaz de tener en cuenta ese tipo de restricciones sin introducir tal redundancia.

Bajo estas premisas, a lo largo de esta disertación se espera demostrar que es posible utilizar la verificación de los conglomerados de componentes para conseguir una detección temprana de defectos. Es muy importante hacer hincapié en que un aspecto fundamental de la tesis es la viabilidad práctica desde el punto de vista de la ingeniería del software; se pretende ofrecer una solución en el plano práctico, problema que deben afrontar muchas técnicas de verificación que no encuentran la difusión deseable por este motivo.

Una vez ubicado el problema, procedería su formulación de manera escueta y precisa. Esto se ha hecho ya en el capítulo 1, y en la página 2 figura la expresión resumida de la tesis; en las páginas que le siguen se describe someramente el sentido de cada expresión implicada en esa frase fundamental. Asimismo, en el capítulo 3.3 se evalúa en detalle cada uno de los aspectos del problema, exponiendo y justificando su interés.

## 3.2. Análisis de las principales técnicas

En el capítulo 2 se ha presentado una revisión de las técnicas que pueden aportar algo a la verificación de sistemas de software basados en componentes. En esta sección se realiza un análisis de las diversas tendencias, con el objeto de señalar en qué medida o aspecto no satisfacen ya la cuestión que da origen a esta tesis.

### 3.2.1. Métodos formales

#### Asequibilidad

El primer problema que pueden plantear los métodos formales de cara a formar parte de la solución de esta tesis es su baja asequibilidad. Los métodos formales han ido conociendo importantes avances en los últimos años; ya no resulta raro que en el desarrollo de software empotrado, en tiempo real, o de componentes críticos intervengan especificaciones formales que persiguen minimizar el número de defectos. Primero la notación Z [Sp89] encontró un grado de difusión respetable, encontrándose actualmente en proceso de estandarización; pero aunque es apropiada para la *especificación*, no lo es tanto para el *desarrollo* de software [Bw97].

Otros métodos, como el Vienna Development Method o VDM [AB92] han encontrado también aceptación en la comunidad de métodos formales. Básicamente, el desarrollo de software mediante un método formal suele presentar los siguientes aspectos:

- Un sistema formal. Una notación formal, precisa, que describe el comportamiento de un sistema y permite demostrar ciertas propiedades.

- Una técnica de desarrollo. Esta técnica suele basarse en el concepto de *refinamiento*: las descripciones formales se van transformando de acuerdo a ciertas reglas, hasta llegar a una descripción lo bastante detallada y cercana a la implementación.
- Una técnica de verificación. Lo que en VDM se llama una "obligación de prueba" consiste en comprobar que la especificación refinada del sistema presenta el mismo comportamiento que la especificación original.

Otras propuestas, como el método B [Ar96], han añadido además métodos orientados al desarrollo de software y herramientas de soporte, que van animando a diferentes organizaciones a incorporar los métodos formales a su proceso productivo (un ejemplo brillante de aplicación de B es el desarrollo del software de frenado del Metro de París). No obstante, multitud de ejemplos dejan claro que esta incorporación parece, cuando menos, problemática.

Según se recoge en [CW97],

> *En el pasado, el uso de métodos formales en la práctica parecía sin esperanzas. Las notaciones eran demasiado oscuras, las técnicas no admitían el aumento de escala, y el soporte de herramientas era inadecuado o bien estas eran demasiado difíciles de utilizar. Sólo había unos pocos casos de estudio no triviales, y no eran lo suficientemente convincentes para el ingeniero de software o hardware de cara a la práctica. Poca gente estaba formada para utilizarlos de manera efectiva en su trabajo. [...] Sólo recientemente hemos empezado a ver un panorama más prometedor para el futuro de los métodos formales.*

Tradicionalmente, pues, parece claro que los métodos formales no eran algo fácil de aplicar. Precisamente, una de las motivaciones de esta tesis es el panorama que se ha descrito en el párrafo anterior: la falta de métodos de especificación lo suficientemente asequibles, lo que en la práctica impide dotar a los componentes software de una descripción sobre sus requisitos y garantías que pueda utilizarse rutinariamente como parte de un proceso de verificación para prevenir defectos. Obsérvese la opinión de David Parnas [Pa96]:

> *Los métodos matemáticos ofrecidos al ingeniero de software en ejercicio no son muy prácticos [...]. La mayoría de ellos, no todos, son correctos desde el punto de vista teórico pero mucho más difíciles de utilizar que las matemáticas que se han desarrollado para el uso en otras ramas de la ingeniería. [...] Necesitamos mucho más trabajo en la notación. La notación que proponen la mayoría de los investigadores en métodos formales es prolija y difícil de leer. Incluso la mejor notación que conozco (la mía, por supuesto) es inadecuada.*

Se recoge esta misma dificultad en [ZW98, pág. 9]; aunque el trabajo de estas autoras va encaminado a la *identificación* de componentes software de cara a la reutilización, y no tanto a la verificación, admiten que "desafortunadamente, por ahora no podemos esperar de los programadores que documenten los componentes de sus programas con especificaciones formales". Su desconfianza respecto de la aplicabilidad generalizada de los métodos formales llega hasta el punto de que su trabajo se enfoca a la identificación de componentes basada en signaturas y no en especificaciones formales, un enfoque que reconocen menos ambicioso pero, a cambio, viable (ya que la información sobre signaturas debe ser suministrada por los programadores en cualquier caso y se obtiene "gratis"). Estas autoras conocen a fondo los métodos formales [Wi95], por lo que está claro que sus dudas no

provienen de la dificultad que ellas mismas puedan encontrar en tales métodos, sino de un análisis bastante objetivo de la situación. En [Wi95] se reconoce también que es frecuente que se formalice sólo lo que es fácil de formalizar.

Larsen et al [LPT94] también trabajan en la dirección de facilitar la incorporación de métodos formales al proceso habitual de producción de software, reconociendo los obstáculos que existen para ello. Bowen y Hinchey, que forman parte del Programming Research Group del Oxford University Computing Laboratory que dio lugar, entre otros logros, a la notación Z, recogen la misma preocupación [BH94]:

> *La comunidad que trabaja en métodos formales es, por lo general, muy buena en lo que se refiere a investigar los aspectos matemáticos de dichos métodos, pero no tanto en promulgar el uso de métodos formales en un entorno de ingeniería y a escala industrial. La transferencia de tecnología es una parte extremadamente importante del esfuerzo general necesario en la aceptación de métodos formales.*

Bowen y Hinchey [BH94] proponen como parte de los "diez mandamientos de los métodos formales" que debe evitarse *sobre-formalizar*. Afirman [pág. 4] que "aplicar métodos formales a todos los aspectos de un sistema sería tanto innecesario como costoso". Incluso en un sistema como el proyecto CICS, por el cual el Oxford University Computing Laboratory e IBM recibieron el Queen's Award for Technological Achievement y que probablemente sea el ejemplo clásico de éxito en la aplicación de métodos formales, sólo alrededor del 10% del sistema fue objeto de desarrollo formal. El desarrollo formal, como tal, se realiza muy rara vez, y la aplicación más frecuente de métodos formales es la *especificación*, que ayuda a detectar ambigüedades o contradicciones.

No obstante, estos y otros autores se apresuran a asegurar que el poco uso de los métodos formales en la industria del desarrollo de software proviene en buena medida de ideas erróneas sobre los mismos [Hl90, BH94b,] y que el uso de métodos formales es perfectamente compatible con la entrega de proyectos en tiempo y coste adecuados (beneficiándola, incluso). Aun pudiendo ser esto cierto, las reservas de la industria hacia los métodos formales (justificadas o no) no dejan de ser un obstáculo de facto, que es necesario tener en cuenta. Basten como ejemplo las reservas que se reflejan en la postura de Pressman [Pr92], que en este caso representa el punto de vista de la ingeniería del software y disfruta de un evidente predicamento y representatividad en este ámbito:

> *La especificación mediante métodos formales es difícil de aprender, representa un "choque cultural" significativo para algunos desarrolladores. Por esta razón, es probable que las técnicas de especificación formales y matemáticas pasen a formar la base de una futura generación de herramientas CASE que utilicen lenguajes de especificación formal como su fundamento.*
>
> *[...] No es realista esperar un uso generalizado de los métodos formales por haber recibido un seminario sobre Z o VDM. El software construido formalmente es software de calidad cuya producción requiere tiempo y pericia.*

David Parnas, que trabaja en el campo de los métodos formales intentando acercarlos a su uso en la industria, reconoce una situación similar [Ei99]:

> *Los estudiantes de Informática de hoy en día aprenden poco de matemáticas fundamentales y los profesionales, a quienes nunca se ha enseñado cómo usarlas, rechazan los métodos matemáticos sin pensárselo dos veces.*

Betrand Meyer, una autoridad en el campo de la programación orientada a objetos (y que trabajó en métodos formales, hasta el punto de haber participado en las primeras versiones de la notación Z) expresa dudas similares respecto a la claridad y facilidad de uso de estas técnicas [Me99, pág. 376]. Clemens Szyperski, en [Sz97], que es la referencia básica por lo que se refiere a componentes software (y recomendado a título personal en correspondencia privada por los editores de WCOP, uno de los más importantes encuentros sobre tecnología de componentes a nivel internacional, como el texto más representativo del estado del arte) afirma [pág. 47] que "dada la complejidad de las especificaciones formales, no es sorprendente que se usen rara vez en la práctica".

**Componentes**

Otro aspecto de esta tesis en el que los métodos formales por sí solos no parecen dar una respuesta adecuada es el enfoque a la combinación de diferentes componentes que mantienen su presencia individual en el sistema. Por supuesto, se contempla la posibilidad de componer especificaciones, y de hecho la reutilización es uno de los "diez mandamientos" citados en [BH94]. En [Sp92] pueden verse también ejemplos de esta composición, pero no se trata exactamente de modelar componentes, sino de realizar operaciones de adición que reuniendo comportamientos parciales conduzcan a la especificación completa. En este párrafo [Sp92, pág. 8] puede verse una descripción del concepto de combinación que se maneja en la notación Z:

> *Hemos especificado los diferentes requisitos de esta operación separadamente, y después los hemos combinado en una sola especificación del comportamiento global de la operación. Esto no significa que cada requisito deba ser implementado separadamente, y las implementaciones combinadas después de alguna forma. De hecho, una implementación podría buscar un lugar para almacenar la nueva fecha, y al mismo tiempo verificar que el nombre no existía previamente; el código para el funcionamiento normal y para el tratamiento de errores podrían estar totalmente mezclados. [...] Los operadores $\vee$ y $\wedge$ no pueden (en general) implementarse eficientemente como formas de combinar programas, pero esto no debe impedirnos usarlos para combinar especificaciones si es conveniente.*

Parece claro, pues, que la combinación de especificaciones no va encaminada a resolver la combinación de elementos software que se plantea en esta tesis. Clarke y Wing [CW97] afirman explícitamente que aún queda trabajo por hacer en el área de la composición: "necesitamos entender cómo componer métodos, componer especificaciones, componer modelos, componer teorías y componer demostraciones".

**Conocimiento**

El planteamiento de esta tesis responde a la conveniencia de aprovechar el conocimiento que se tiene sobre los componentes que forman parte de un sistema, como forma de prevenir defectos en el sistema resultante. Evidentemente, el conocimiento se puede plasmar de muy diversas formas y una notación formal permite también hacerlo; pero los

argumentos ofrecidos respecto a lo poco asequible de las notaciones formales apoyan también la idea de que existen formas de plasmar el conocimiento mucho más naturales, al menos para usos genéricos. Las notaciones formales están orientadas hacia la descripción de requisitos *funcionales*; no obstante, existen multitud de requisitos no-funcionales cuya no satisfacción puede provocar la caída del sistema (para Szyperski [Sz97, pág. 46] "resulta obvio que en la mayoría de los casos prácticos una violación de requisitos no funcionales puede hacer fallar a los clientes tan fácilmente como una violación de los requisitos funcionales"). Por ejemplo [Sz97, pág. 44] puede darse el caso de que un paquete de animación se base en una biblioteca de funciones matemáticas; una simple "mejora" en la precisión de los resultados que ofrece la biblioteca puede beneficiar a ciertos clientes de la misma, pero si esa mejora de precisión se hace a costa de la velocidad, el paquete de animación puede quedar inservible, al ser incapaz de suministrar cuadros con la frecuencia adecuada. Una notación formal se encuentra con ciertas limitaciones para representar este tipo de requisitos.

Como se afirma en [BH95], es importante, además, elegir una notación bien asentada y con una buena base de usuarios si se quiere que se aplique con éxito en la práctica. Es habitual que en el desarrollo de una notación formal para la industria pasen al menos diez años desde la concepción hasta la aplicación real; la transferencia tecnológica está, además, llena de obstáculos. Por eso parece necesario pensar en una solución más adecuada en ese aspecto, cosa que se ha intentado hacer en esta tesis.

**Automatización**

Por lo que se refiere a la automatización, las demostraciones automatizadas ya encuentran soporte en herramientas, pero son algo relativamente reciente [BH94]. En [CW97, pág. 13] se recoge también la necesidad, para el futuro, de mejorar la facilidad de uso de las herramientas para métodos formales.

**Flexibilidad**

La aplicación de métodos formales a muy diferentes ámbitos del proceso de desarrollo no parece sencilla. Bowen y Hinchey [BH94] afirman que siempre existe tensión entre la expresividad de un lenguaje y los niveles de abstracción que soporta. Los lenguajes con "vocabularios pequeños" ofrecen niveles de abstracción mayores. Debe elegirse la notación formal apropiada para el problema apropiado; puesto que la cuestión que se plantea en esta tesis requiere de un grado máximo de generalidad, puede pensarse que las notaciones formales al uso no resultarán adecuadas en muchos de los posibles campos de aplicación (muchas veces no se necesita la potencia de una notación formal, sino una forma simple de plasmar conocimiento, que en muchos casos ni siquiera se refiere a funcionalidad). Así lo confirman las impresiones de Shaw y Garlan [SG95]:

> *El diseño arquitectónico depende fuertemente de especificaciones precisas de los subsistemas y sus interacciones. Estas especificaciones deben cubrir un amplio abanico de propiedades, así que las notaciones de especificación –y los métodos asociados- deben seleccionarse o desarrollarse de modo que las propiedades interesantes encajen. Desgraciadamente, los métodos formales disponibles sólo encajan parcialmente con las necesidades arquitectónicas, que conllevan descripciones de estructura, empaquetado, suposiciones sobre el entorno, representación y eficiencia además de funcionalidad.*

Por tanto, en este aspecto los métodos formales tampoco parecen ser una respuesta a la cuestión que se plantea en esta tesis.

**Conclusión**

Aunque la difusión y aplicación práctica de los métodos formales está mejorando, estos métodos por sí solos no parecen dar adecuada respuesta a la pregunta que se desarrolla aquí. Son tradicionalmente difíciles de aplicar (y aun cuando esto vaya cambiando paulatinamente, la imagen que se tiene de ellos es un inconveniente en cualquier caso), no están orientados al tratamiento de componentes, no parecen ser la notación más adecuada cuando el énfasis se pone en plasmar conocimiento y no en un alto grado de formalismo, el soporte de herramientas para automatizar su tratamiento está aún bajo desarrollo, y no resultan lo suficientemente flexibles.

### 3.2.2. Análisis estático e interpretación abstracta

Aun habiéndose desarrollado bases teóricas para la interpretación abstracta hace ya más de veinte años, muy pocos proyectos incorporan estas ideas (buena parte de las aplicaciones prácticas de análisis estático están encaminadas a la optimización de código en compiladores, y no a la detección temprana de defectos en programas). La interpretación abstracta no ha encontrado un uso generalizado en la industria del desarrollo de software. Estas técnicas [CC01, p. 1] se hallan en estos momentos en una fase de industrialización.

Respecto a su relación con los componentes software, muchos de los métodos de interpretación abstracta son *globales* en el sentido de que necesitan evaluar el código fuente completo del programa. En 2.3 se ha mencionado ya que existen tendencias en la línea de acercar la interpretación abstracta y las tecnologías de componentes software; no obstante, en ese mismo capítulo se ve que la motivación no coincide con la expresada aquí. En [CC01], por ejemplo, se presenta el paradigma de componentes como un mero recurso para resolver un problema distinto, y no como un fin en sí mismo. Los autores plantean el problema del análisis preciso de grandes programas, y ofrecen como solución una interpretación *local*, parcial, sobre fragmentos pequeños (lo que disminuye las exigencias de recursos de tiempo y espacio en el cómputo) que mediante diversas técnicas posibles se combinaría después. Esos ámbitos parciales pueden ser componentes software, pero también clases, módulos o cualquier otra construcción que permita limitar el tamaño. En este planteamiento los componentes son, pues, algo puramente accidental.

Hay otra reflexión que hacer a este respecto: en ocasiones, lo que se desea verificar en un sistema no es su comportamiento a un nivel algorítmico riguroso. Frecuentemente, los programadores o diseñadores tienen un conocimiento sobre el sistema que podría ser aprovechado sin un análisis del tipo descrito; es evidente que muchos programadores son capaces de afirmar que un programa dado contiene un error o que se bloqueará, aunque no sean capaces de demostrarlo algorítmicamente. Si todo el conocimiento que programadores, diseñadores, fabricantes, etc. tienen sobre los componentes software pudiera describirse formalmente y de manera sencilla, sería posible generar una "base de conocimientos" que describe todo lo que se sabe sobre el sistema que se está construyendo; esto permitiría verificar si los requisitos de ciertos componentes se cumplen o no, y por tanto prever defectos de funcionamiento. Un sistema de inferencia convencional, basado en lógica de primer orden, podría realizar estas verificaciones con relativa facilidad.

En cualquier caso, la comprensión de la base teórica de estas técnicas plantea muy notables exigencias formativas. Los fundamentos del análisis estático conllevan una fuerte base matemática y formal; como ejemplo, el propio François Thomasset [Thom], un investigador en el campo de la interpretación abstracta enfocada a la optimización de código, reconoce que no le resulta fácil leer los trabajos de Patrick Cousot [Th00]. Aunque las herramientas oculten esta complejidad a los usuarios, ciertamente el desarrollo o adaptación de algoritmos de análisis estático para ajustarse a necesidades específicas no parece una tarea que cualquier equipo de desarrollo pueda abordar a un coste razonable.

Además, los algoritmos de análisis estático son bastante específicos, como se puede deducir de los casos presentados en 2.3. La interpretación abstracta no es una técnica abierta, flexible, general, como se plantea en los requisitos de esta tesis. Hay mucha información sobre los componentes software que no parece que pueda plasmarse o analizarse mediante técnicas de interpretación abstracta. Por ejemplo, PolySpace Verifier es capaz de detectar ciertos tipos de problemas, como se mostró en 2.3, pero no hay opción para su adaptación a otros tipos de verificaciones. Otro factor notable es que el verificador se basa en analizar código fuente, lo que resulta demasiado restrictivo para dar respuesta a esta tesis. Además, debe contar con un analizador para el lenguaje específico manejado; si el proyecto desarrollado está en Pascal, Smalltalk o Java, por ejemplo, no se puede utilizar PolySpace Verifier tal y como se comercializa actualmente.

Como conclusión, las técnicas de interpretación abstracta no tienen una relación estrecha con los componentes software de manera específica, no resultan fáciles de desarrollar por personal típico de desarrollo sin una formación previa (que posiblemente involucraría una fuerte base matemática), y no resultan particularmente flexibles en su aplicación. Por tanto, puede decirse que aunque hay una coincidencia parcial de objetivos, no resultan técnicas apropiadas para nuestros propósitos y no dan respuesta a los planteamientos de esta tesis.

### 3.2.3. Técnicas de especificación semántica

De cara a los objetivos de esta tesis, las técnicas tradicionales de especificación semántica de lenguajes adolecen de un reconocido problema de modularidad y legibilidad [Wt96]. En general puede hablarse de los mismos problemas de asequibilidad que presentan otros métodos formales, como se ha descrito en detalle en el capítulo 3.2.1, requiriendo una cierta mentalidad matemática y seguramente una formación específica para su aplicación.

Por otra parte, la modularidad también es un punto débil, aunque los trabajos de semántica monádica reutilizable citados anteriormente van encaminados precisamente a mejorar este aspecto.

Si se superasen estas limitaciones de legibilidad y modularidad, quizás estas técnicas tendrían aplicación para resolver el problema que nos ocupa bajo restricciones de viabilidad razonables; pero lo cierto es que en su estado actual no parecen utilizables para la verificación de componentes del tipo que aquí se propone. Existen escasas implementaciones basadas en estas técnicas, y aunque constituyen una prometedora línea de futuro, no se encuentran en un estado de madurez (ni de adecuación al problema de los componentes) que permita su uso práctico para nuestros propósitos.

### 3.2.4. Especificación de procesos

En una primera aproximación, puede pensarse que la especificación de procesos es un campo en el que cabe encontrar similitudes con lo que se plantea en esta disertación. El modelo de procesos que se comunican constituye la base de CSP, y ha influido decisivamente en el π-cálculo y a través de él en otras muchas iniciativas; en términos generales, este modelo presenta similitudes con el modelo de Itacio.

Se plantea la existencia de entidades fuertemente encapsuladas –los procesos- que se comunican con el exterior a través de sus canales de comunicación, y además dichas entidades pueden a su vez estar constituidas por más entidades análogas, de nivel inferior, con conexiones privadas. Estas ideas no difieren demasiado de la concepción estructural de Itacio respecto a los componentes. No es descabellado pensar que sea posible realizar una correspondencia entre ambos modelos, a través de una instanciación específica del modelo Itacio.

No obstante, las corrientes englobadas en el capítulo 2.5 no dan respuesta, tal y como se formulan, a las exigencias planteadas en esta tesis, por las razones que se evalúan a continuación.

#### CSP y otras álgebras de procesos

CSP es, como se ha dicho, una notación formal para describir los procesos y su interacción; se trata de un caso de *álgebra de procesos*. En primer lugar, cabe decir –y así puede apreciarse repetidamente en la literatura sobre CSP- que esta notación está fuertemente orientada a la descripción de procesos. Ciertamente, CSP muestra un enfoque ambicioso respecto a los procesos; se ha planteado la idea de que cualquier computación pueda modelarse como un proceso, y desde este punto de vista CSP sería un modelo generalista, al menos en un plano teórico. Pero lo cierto es que el propósito principal de CSP es abordar sobre todo problemas en los que esté presente la concurrencia y el paralelismo. Por tanto, aunque fuese una técnica aplicable de manera indirecta al terreno de los componentes software, no resulta evidente cómo podría lograrse tal cosa.

La aplicabilidad choca con otra barrera, ya evaluada en otros puntos de esta disertación. CSP es una notación formal, con las dificultades de adopción que esto suele conllevar en muchas organizaciones de desarrollo. No resulta, probablemente, tan abstracta como otras técnicas formales mencionadas aquí, pero eso es precisamente porque es una técnica menos general y más orientada a la idea de procesos, con lo que se estaría ganando en facilidad de uso frente a notaciones como Z pero perdiendo en flexibilidad.

La orientación de CSP al modelado de procesos hace que también su utilidad de cara a la verificación –en el sentido planteado en la tesis que nos ocupa- parezca muy limitada. Ciertamente, CSP y otras álgebras de procesos son construcciones rigurosas y formales, lo que da pie a su automatización, y de hecho sí se pueden realizar ciertos tipos de verificaciones estáticas en los sistemas, pero no tan flexibles y adaptables como se plantea como requisito en esta tesis.

#### π-cálculo y π$\mathcal{L}$-cálculo

El π-cálculo y el π$\mathcal{L}$-cálculo son cálculos de procesos móviles, derivados del álgebra de procesos CCS. De cara a los requisitos de esta tesis, pueden señalarse problemas similares a

los descritos para las álgebras de procesos. π-cálculo y π*L*-cálculo nacieron como una herramienta conceptual para razonar sobre concurrencia y paralelismo, y pretenden ser un fundamento teórico para lenguajes de programación concurrente; no son apropiados, por tanto, para modelar de manera directa sistemas basados en componentes, ni parecen lo suficientemente flexibles para los problemas de desarrollo que se pretende resolver aquí. En muchos casos, los sistemas basados en componentes no implican concurrencia, y la identificación entre el concepto de proceso y el de componente puede resultar poco natural.

Además, estos artefactos se revelan como de muy bajo nivel de cara a su uso práctico (como se cita en 2.5.3 y al hablar de Pict en 2.5.4), lo que en cualquier caso exige algún tipo de desarrollo adicional. La sólida base teórica que estos modelos ofrecen puede servir como un apoyo prometedor de cara a desarrollar modelos de componentes, pero tal como están formulados no dan respuesta a lo planteado para esta tesis.

**Lenguajes derivados**

Los lenguajes derivados (directa o indirectamente) de las álgebras de procesos o el π-cálculo tienen algunas ventajas. Su viabilidad práctica es mucho mayor; de hecho, estos lenguajes tienen implementaciones, más o menos experimentales, que demuestran que se hallan mucho más cerca de la etapa de producción que las teorías que les sirven de base (cosa perfectamente lógica, ya que dichas teorías son precisamente eso, desarrollos teóricos fundamentales).

Además, estos lenguajes adoptan la forma de lenguajes de programación; aunque sean lenguajes con ciertas peculiaridades (como un enfoque fuertemente funcional, por ejemplo) su comprensión y uso no parece fuera del alcance de los desarrolladores de software típicos. Parece claro, asimismo, que resultarían mucho más comprensibles al profano que sus equivalentes teóricos.

Eso no basta para dar respuesta a los planteamientos de esta tesis. Algunos de estos lenguajes (como Pict) están también muy influidos por el paradigma de la programación concurrente, y no están dirigidos a la composición de componentes. Otros no presentan ese problema; Piccola u otros lenguajes de composición se han diseñado explícitamente para abordar el desarrollo de sistemas basados en componentes, aunque su origen teórico se halle en el π*L*-cálculo. Pero el tipo de problemas que Piccola y similares pretenden resolver no es el planteado aquí; se centran más bien en la composición, en la **construcción** del software, y no tanto en la verificación de los requisitos de los diversos componentes.

En general, estos lenguajes de composición adoptan un enfoque funcional en el sentido de que pretenden llevar a cabo la funcionalidad del sistema sirviendo como *pegamento* entre los diversos componentes. Piccola, por ejemplo [Ac99] proporciona abstracciones para componer componentes, invocar servicios y definir nuevos servicios.

Su principal motivación no es la verificación cruzada de los requisitos de los componentes involucrados. Esto no imposibilita su uso o ampliación para los fines de verificación estática que aquí se persiguen, pero lo cierto es que no se han diseñado con dichos fines como motivo principal, y por tanto tampoco dan una respuesta adecuada a esta tesis. En [ALS00, p. 14] se presenta un ejemplo de incoherencia entre componentes en Piccola, pero se detecta "porque sí"; las explicaciones se centran en construir un adaptador para *solucionar* el problema, no en *detectarlo*. En [Ac99] se ofrece una lista de dominios en los que el uso de Piccola puede ser útil. Resulta muy flexible en el sentido de que no se adhiere a un estilo

arquitectónico específico, y ofrece soporte para varios estilos diferentes; pero no se menciona preocupación especial por la verificación. Se entiende una interfaz como un simple conjunto de servicios bajo un nombre, sin incluir referencia alguna a restricciones asociadas. Cuando se habla de evitar conexiones no válidas, se confía para ello en que los conectores descritos en los diversos estilos arquitectónicos garantizarán la corrección por construcción, y se citan los mismos trabajos que hemos citado aquí respecto a los estilos arquitectónicos. Estas ideas se analizan a la luz de nuestros requisitos en otros puntos de esta disertación, y se señalan sus limitaciones.

Siguiendo con Piccola, también en [Ac99, p. 7] se habla de modelos de componentes y también de contratos (haciendo referencia, por cierto, a casi los mismos tipos de contrato que se han evaluado aquí). Se explica que en Piccola se pueden expresar algunos de estos contratos, y que otros "posiblemente se podrían derivar usando la semántica de Piccola", pero esto se plantea casi como una línea de investigación futura y no como una motivación básica (además de que no se ofrecen detalles sobre qué puede expresarse y cómo, o qué no puede expresarse). Se trata en realidad de preguntas abiertas, sobre las cuales se puede razonar con el modelo de Piccola, pero que no están resueltas por el lenguaje.

Como conclusión, parece evidente que aunque en estos lenguajes se plantea en algún momento el problema de la verificación no constituye una motivación básica, y se trata de una cuestión abierta. Las técnicas en las que se basan, además, se hallan aún bajo desarrollo.

### 3.2.5. Programación por contratos

#### Meyer

Aunque a primera vista puede parecer que la programación por contratos ya satisface básicamente los interrogantes que se plantean en esta tesis, lo cierto es que existen importantes diferencias entre dicho esquema de verificación y el propuesto aquí. A nuestro juicio, la programación por contratos propuesta por Meyer resulta claramente recomendable de cara a satisfacer ciertos objetivos, pero presenta importantes lagunas en relación con los objetivos de esta tesis.

En primer lugar, Meyer persigue realizar una especificación del software (es decir, una descripción de "qué se entiende por comportamiento correcto" para cierto elemento de software). Para ello, se basa en una lógica de predicados sobre el *estado concreto* del elemento a verificar; la propuesta de Meyer es, dicho de forma simple, expresar los contratos mediante ciertos recursos del propio lenguaje de programación, ciertas expresiones que se ejecutan imbricadas con la propia implementación de la funcionalidad. Esto es así porque:

- Meyer encuentra ventajas en esta comprobación "permanente", que permite saber que el sistema *está funcionando* de acuerdo con las especificaciones.
- Por otra parte, reconoce que aunque el candidato natural para estas especificaciones formales sería la lógica de predicados de primer orden [Me99, pág. 378] esto no le permitiría alcanzar su propósito de verificar el código fuente.

De cara a nuestros objetivos, el construir las especificaciones con elementos del lenguaje de programación (incluyendo funciones imperativas) va contra la idea fundamental del análisis estático que aquí planteamos.

Evidentemente, las expresiones asertivas *ejecutables* que Meyer propone no se pueden analizar de manera estática (a menos que se añada una nueva capa de interpretación abstracta sobre las mismas, tarea nada sencilla y que se comenta en otros puntos de esta tesis). Empeora esto el hecho de que la verificación de precondiciones, postcondiciones e invariantes puede ser costosa en tiempo de ejecución, como ya se ha dicho, y frecuentemente se desactiva al generar las versiones definitivas de los programas. Al fin y al cabo, se sigue dependiendo de una fase de pruebas, y los casos de prueba previstos pueden no forzar ninguna ruptura de ningún contrato o invariante, mientras que tal situación puede darse cuando el sistema ya está en marcha... con el agravante de que entonces probablemente estén desactivadas tales comprobaciones.

En [Me99, pág. 323] se explica básicamente que lo que una clase dice a sus clientes es:

> *"Si usted me promete llamar al método con las precondiciones satisfechas, entonces yo le prometo entregar un estado final en el que las postcondiciones están satisfechas".*

Lo cierto es que esa expresión no es exacta. Los contratos de Meyer no se basan en promesas, sino en hechos; no se basan en el *estado posible o abstracto*, sino en *estados concretos* del sistema. Es por esto también que las pre/postcondiciones son código ejecutable del lenguaje Eiffel; no tiene sentido su cálculo si no se refieren a un objeto concreto con un estado concreto. Por tanto, a la clase no le importan las *promesas* de su cliente; lo único que ocurre es que, si en la práctica se da el caso de incumplimiento de las precondiciones, la clase tendrá la libertad de incumplir sus postcondiciones. Es perfectamente posible conectar a un objeto servidor con un objeto cliente que no cumpla las precondiciones de dicho servidor, o que no las cumpla siempre. Simplemente, en tiempo de ejecución se producirá un fallo que, eso sí, el sistema detectará (insisto, siempre y cuando las verificaciones no estén desactivadas, cosa frecuente).

Por tanto, está claro que este sistema de verificación no permite realizar un análisis estático, y menos aún un análisis estático automatizado. Su objetivo no es, ni mucho menos, ese. Esta es una diferencia radical con lo propuesto en esta tesis.

Otros aspectos de los contratos de Meyer sí serían adecuados para responder satisfactoriamente la pregunta que da origen a esta tesis; su notación permite representar conocimiento, y permite hacerlo además de forma asequible para el perfil de un desarrollador (acostumbrado a expresarse en lenguajes de programación). Pero nuevamente choca con las ideas de flexibilidad que aquí se propugnan; si no estamos hablando de clases, sino que los componentes manejados son componentes en la acepción habitual de la palabra (módulos binarios), u otro tipo de elementos, este tipo de contratos podría no tener aplicación, puesto que están ligados a un nivel de abstracción muy específico.

Por el contrario, Meyer rechaza la lógica de predicados precisamente pensando en su objetivo específico de verificar a nivel de código ejecutable [Me99, pág. 378]:

> *Tomemos lo que la mayoría de la gente ducha en el tema ha sugerido como el candidato natural: el cálculo de predicados de primer orden. Este formalismo no nos capacita para expresar algunas propiedades de interés inmediato para los desarrolladores y de uso común en las aserciones, tal como "el grafo no tiene bucles" (una cláusula invariante típica).*

Acto seguido, se contrapone la facilidad con la que un programador puede escribir una rutina que verifique tal cosa, lo que lleva a justificar (acertadamente) el modelo de contratos de Meyer. La cuestión es que este argumento es válido para los objetivos que pretende alcanzar este modelo de contratos, pero no para los de esta tesis. Como se ha dicho ya –y se verá en su momento- uno de los propósitos del modelo aquí propuesto es ofrecer un marco en el que incorporar conocimiento. La afirmación "el grafo no tiene bucles" se puede expresar de manera trivial en lógica de primer orden, y eso puede ser suficiente **para ciertos propósitos**; no para verificar el comportamiento del sistema en el plano imperativo, que es lo que persigue Meyer, pero quizás sí para detectar incoherencias entre las especificaciones de diversos componentes. Cuando se da una situación de este tipo, es decir, cuando existen partes de la especificación que no se pueden expresar con construcciones del lenguaje, la programación por contratos se limita a describir estos requisitos mediante cláusulas informales. Como dice Szyperski [Sz97, pág. 46] las pre/postcondiciones están ampliamente aceptadas y se suelen combinar con cláusulas informales para completar los contratos, pero el inconveniente de esto es que queda excluida la verificación formal o automatizada. Por tanto, el modelo de programación por contratos de Meyer resulta apropiado –y en algunos sentidos insustituible- en ciertos ámbitos, pero queda claro que no da respuesta a la cuestión planteada en esta tesis.

**Contratos bilaterales: Wirfs-Brock, Reenskaug et al**

En este caso se citan los contratos en la versión de Wirfs-Brock, Trygve Reenskaug y otros con el objeto de completar la revisión del estado del arte presentado, pero ciertamente esta línea de investigación se aleja notablemente de la motivación de esta tesis. No en un sentido abstracto, ya que recogen la preocupación por la interacción entre componentes –en este caso objetos o roles-; pero sí en un sentido concreto, ya que estos autores centran sus esfuerzos en técnicas de análisis y modelado, con lo que quedarían fuera de lugar consideraciones fundamentales en esta tesis, tales como la verificación, la posibilidad de automatizar el proceso de verificación, etc. Está claro que los trabajos mencionados tienen bastante relación con el tema que nos ocupa y una evidente influencia en el *estado del arte*, pero de ninguna manera resuelven el problema planteado.

**Contratos de reutilización (Vrije Universiteit Brussel)**

Los contratos de reutilización propuestos por el grupo PTL-VUB constituyen una interesante aproximación al problema planteado en esta tesis. De alguna forma, recogen el "estilo general" de lo que se pretende conseguir aquí, si se piensa en los componentes como objetos o como participantes de un contrato. Ciertamente mediante esta técnica se pueden plasmar ciertas formas de conocimiento sobre la relación entre esos "componentes". Este tipo de contratos, además, permite realizar algunas formas de verificación estática, sobre todo en lo referente a la estructura correcta de los contratos y a su evolución temporal mediante la aplicación de los mencionados operadores. Esta verificación es además susceptible de ser automatizada hasta cierto punto, y su relativa simplicidad hace que resulte también razonablemente asequible, al menos en lo que se refiere a la formación necesaria.

No obstante, también se encuentra bastante lejos de dar solución al planteamiento de esta tesis. Un primer problema es que el conocimiento que se puede plasmar mediante los contratos de reutilización es muy limitado. Si los componentes que se manejan no encajan en el papel de "objetos" o participantes –por ejemplo si se está hablando de bibliotecas

completas- puede no tener demasiado sentido el uso de contratos de reutilización. Y aunque se esté tratando con clases, los contratos de reutilización permiten plasmar sólo aspectos muy concretos de su interacción. Por ejemplo, muchos requisitos no funcionales de los mencionados en otros puntos de esta tesis no podrían ser adecuadamente reflejados en un contrato de reutilización. Por su parte, el propio proceso de verificación (tal como se plantea, por ejemplo, en [Lu97]) se ha construido prácticamente *ad hoc*.

Esta dificultad para reflejar conocimiento influye también en el apartado de asequibilidad: uno de los requisitos planteados en este sentido era la flexibilidad y generalidad, de las que estos contratos claramente carecen, al cumplir una misión bastante específica.

Por tanto, aunque se trate de un método comprensible y automatizable deja sin solución buena parte de nuestros interrogantes: no es general, no está orientado a componentes (sino a objetos o, como mucho, roles), y no permite almacenar ciertas formas de conocimiento sobre los componentes (menos aún verificarlas automáticamente).

**Lenguaje Contract (Northeastern University)**

De cara a los objetivos de esta tesis, los trabajos del CSS-NEU respecto al lenguaje Contract presentan un cierto interés, aunque también quedan lejos de solucionar los problemas planteados.

La visión del término "contrato" aportada por el lenguaje Contract refleja el interés por describir el comportamiento de un componente –en este caso un objeto- en relación con los demás, así como por documentar dicho comportamiento. En este sentido, va en la dirección propuesta en esta tesis.

Sin embargo, en absoluto responde a la cuestión planteada. El lenguaje Contract está orientado a la representación de contratos y a la reutilización de los esquemas parciales de interacción, pero no a la verificación de los mismos. Además, dicha verificación parece cuando menos problemática, ya que la forma de describir los métodos e interfaces en Contract no difiere mucho de la forma en que se hace cuando se utiliza un lenguaje de programación. Eso nos llevaría nuevamente a técnicas de interpretación abstracta o similares, relativamente complejas.

### 3.2.6. Estilos arquitectónicos e incoherencias

En general, los trabajos sobre estilos arquitectónicos constituyen una aproximación interesante a un problema similar al planteado aquí. La combinación de partes que responden a diferentes estilos arquitectónicos no deja de ser también un conglomerado de componentes, y la detección de incoherencias entre esas partes responde a la misma preocupación que esta tesis: la necesidad de prevenir los *errores emergentes* que aparecen en un sistema y que no son fruto de defectos intrínsecos de ninguna de sus partes, sino de defectos consecuencia de la propia combinación de las mismas.

De cara a la cuestión base de esta tesis, uno de los problemas de este punto de vista es la rigidez en cuanto a los niveles de abstracción. Una de las fuentes de Gacek es un trabajo de Kazman et al [KCAB97] y citándolo Gacek afirma que la clasificación de estilos y características que propone es aplicable en etapas más tempranas porque se mueve en un plano más abstracto (sistemas) que el de Kazman (elementos). En esta tesis se pretende precisamente partir de un campo de actuación menos abstracto aún que el de Kazman et al,

aportando técnicas que reduzcan los errores que se producen en tareas de diseño avanzado, implementación e incluso distribución. El uso de los estilos arquitectónicos va dirigido a "un nivel de granularidad grueso", en palabras de Abd-Allah y Boehm, que explican que la aparición de arquitecturas de software y estilos arquitectónicos plantea la composición de sistemas a un nivel muy por encima del de los lenguajes de programación [AB95, p. 1]. Esto limita enormemente su aplicación de cara a otros niveles de abstracción como clases/objetos, o incluso compromete su aplicabilidad en componentes software comerciales, si se trata de componentes de tamaño relativamente *pequeño*. Como ejemplo de esto, los estilos arquitectónicos mencionados por Garlan y Shaw en [GS93] son los siguientes:

- Tubos (*pipes*) y filtros
- Orientado a objetos
- Programa principal / subrutinas
- Repositorio
- Basado en eventos (invocación implícita)
- Basado en reglas
- Basado en transición de estados
- Control de procesos (retroalimentación)
- Específico del dominio
- Heterogéneo

Parece claro que muchas características interesantes de los componentes (en su acepción más extendida) no pueden encuadrarse en el marco conceptual de los estilos arquitectónicos. En [AB95] se menciona también que "los estilos arquitectónicos son un medio potente para comprender el diseño de grandes sistemas de software", quedando implícita la orientación de esta disciplina hacia sistemas de gran tamaño. Medvidovic y Taylor [MT98] insisten en que los trabajos sobre arquitecturas tiene aplicación en la evolución de sistemas *de grano grueso* [p. 106] y no resultan apropiados para la evolución *de grano fino* [p. 107].

Sí parece posible que la combinación de estilos arquitectónicos se realice de forma estática, y de hecho ese es probablemente su único campo de actuación, puesto que están relacionados con el diseño a alto nivel. La automatización de este proceso resulta comprometida, puesto que los diferentes estilos arquitectónicos poseen no ya valores diferentes de las mismas características, sino características de hecho distintas; no obstante, los trabajos realizados en este campo van precisamente en la línea de una formalización y homogeneización de estos estilos arquitectónicos, que permita su comparación e integración de manera relativamente formal.

Respecto a la viabilidad o facilidad de implantación de estas ideas, también surgen dudas precisamente por la limitación de aplicabilidad: cualquier proyecto de desarrollo que no involucre la combinación de sistemas de tamaño relativamente grande no encontraría ventajas significativas en el uso de estas técnicas, y aun en el caso de que fuesen aplicables por tratarse de un proyecto de integración a gran escala, serían apropiadas para razonar

sobre problemas muy concretos del diseño de arquitectura de muy alto nivel, dejando fuera un enorme porcentaje del proceso de desarrollo.

## 3.2.7. Lenguajes de Descripción de Arquitecturas (ADL)

### Coincidencia parcial de objetivos

Los ADL pueden parecer opciones atractivas para nuestros objetivos, pero esto nos ofrecerá sólo ventajas parciales. Por ejemplo, al evaluar Darwin puede pensarse que las ideas de componentes que ofrecen y requieren servicios, o la noción de *independencia del contexto*, o la múltiple instanciación de la descripción de un componente, se acercan al modelo de componentes que aquí se pretende delinear (lo mismo vale para UniCon y otros). Se pueden citar similitudes parciales como estas, pero en términos globales los ADL no resultan satisfactorios para nuestro propósito. En primer lugar hay que señalar que muchos de estos lenguajes están orientados a la descripción de sistemas concurrentes, como es el caso de Darwin o WRIGHT. Ese objetivo básico no tiene mucha relación con esta tesis. En general, existe un solapamiento de objetivos entre Itacio y estos lenguajes, pero muy leve; véase en la pág. 39 la lista de beneficios esperados para UniCon y técnicas asociadas. Existe algún grado de coincidencia (por ejemplo, en el punto 3) pero claramente el énfasis no está en las áreas de interés de esta tesis.

### Análisis estático

Ya se ha mencionado que por lo general los ADL no están directamente enfocados a la realización de un análisis estático; su principal motivación suele ser la capacidad de especificación de la arquitectura de un sistema, evitando el excesivo esfuerzo de desarrollo que requiere el uso de métodos formales de propósito general. Los ADL proporcionan un nivel de abstracción y unas construcciones básicas adecuadas a la descripción de arquitectura, pero no resultan tan útiles de cara al tipo de verificaciones estáticas que aquí se pretende realizar. Está claro, no obstante, que aunque el análisis estático no sea un objetivo central sí se ha tenido en cuenta esta posibilidad, y se han desarrollado trabajos en ese sentido [NACO97]. Merece la pena evaluar algunos ADL que ofrecen vías para el análisis estático, como es el caso de WRIGHT. Se verá que, por lo general, los objetivos y características de estos ADL no los hacen apropiados para responder a las exigencias de esta tesis.

Las restricciones que se utilizan, por ejemplo, en WRIGHT, hacen referencia a elementos de la arquitectura (un conjunto determinado de los mismos), con el propósito de verificar características globales de los sistemas. Esto entronca con problemas ya mencionados: el nivel de abstracción parece claramente superior al que en ocasiones interesa para los propósitos de Itacio. Desde este punto de vista, las restricciones de WRIGHT y las técnicas que propone podrían ser complementarias al uso de Itacio, pero no persiguen el mismo uso.

En [Al97] se examinan las particularidades de los análisis de completud y consistencia. El análisis de consistencia, que persigue encontrar contradicciones en el sistema, encajaría en parte con los propósitos de esta tesis, pero en el caso de WRIGHT se entiende principalmente como un análisis previo, que permita realizar los demás procesos de razonamiento con corrección, y no como un fin en sí mismo. Además, el análisis de consistencia aparece claramente restringido, ya que se enumeran 11 verificaciones concretas

(siempre referidas a las entidades que se manejan en WRIGHT y no tanto a *qué representan* esas entidades en un momento dado).

En el caso de Rapide, también se plantean diversos tipos de verificación estática, pero ligados a la idea de los patrones de eventos y la programación de sistemas concurrentes, lo que limita su aplicabilidad a ciertos tipos de problemas.

**Flexibilidad**

El tipo de análisis estático que puede realizarse en lenguajes como WRIGHT, además, no resulta lo bastante flexible para nuestros propósitos. En [Al97, p. 66 y siguientes], se habla de las posibilidades que ofrece el análisis de especificaciones de WRIGHT:

> *Hay muchos tipos de análisis que se pueden considerar al nivel arquitectónico de diseño. Estos análisis incluyen la corrección funcional del sistema, el potencial de expansión para acomodar exigencias crecientes, la contención para recursos críticos, y el coste probable de implementación de subsistemas, entre muchos otros. Cada uno de estos análisis se apoya en diferentes propiedades del sistema descrito, y lo propio sería que fuese soportado por diferentes formalismos arquitectónicos.*

La primera frase de esta cita ahonda en la idea mencionada previamente, que los lenguajes de descripción arquitectónica van encaminados, precisamente, a descripciones a nivel de arquitectura, aunque se puedan utilizar a diferentes niveles de abstracción mediante el anidamiento. Aparte de eso, queda claro que el autor perfila posibles tipos de análisis, basados en variables específicas, y que requerirían formalismos específicos.

Esta idea se ve apoyada por otros autores. Robbins et al [RMRR98] afirman que parte de la comunidad que investiga en arquitecturas software, sobre todo personal académico, se ha centrado en la evaluación analítica de descripciones arquitectónicas, y que para dar respuesta a preguntas de difícil evaluación es necesario disponer de técnicas potentes de modelado y análisis que contemplen en profundidad aspectos específicos. Por ejemplo, determinar la posibilidad de interbloqueo requiere modelos formales especializados del comportamiento y comunicación de los hilos; como contrapartida, el rigor de los métodos formales aleja la atención del desarrollador de las cuestiones "cotidianas" del desarrollo. El uso de lenguajes de modelado de propósito específico ha fragmentado notablemente la comunidad que se dedica a la investigación de la evaluación analítica de arquitecturas.

Esta especificidad, en parte, aleja WRIGHT y lenguajes similares de nuestros objetivos de viabilidad. Las posibilidades de análisis estático de un ADL estarían limitadas de antemano a ciertos tipos de comprobaciones, y el desarrollo de verificaciones sobre aspectos completamente nuevos podría requerir una ampliación de la teoría en la que se basa el ADL, lo que claramente no es operativo. Nuestra tesis plantea la necesidad de un sistema de verificación homogéneo, abierto, cuya aplicación resulte rentable precisamente por su simplicidad y facilidad de explotación según las necesidades que surjan en el desarrollo. Si se examina el modelo de UniCon, por poner otro ejemplo, se verá que los tipos de entidades que se manejan están claramente establecidos, y esto puede resultar un factor limitador para aplicaciones del modelo en otros niveles de abstracción.

**Notación y asequibilidad**

En algunos ADLs existen también problemas de notación (de cara a ser utilizados en esta tesis, se entiende). La notación utilizada en WRIGHT se basa en CSP, y en concreto las verificaciones de consistencia se apoyan en el concepto de refinamiento de procesos de CSP, lo que añade un nuevo ingrediente de complejidad a WRIGHT. En otros puntos de este documento (véase 3.2.1) se mencionan algunos de los problemas que plantean los métodos formales en general, y la notación de WRIGHT puede considerarse afectada por los mismos problemas que otros métodos formales.

**Conclusión**

Como conclusión, los lenguajes de descripción de arquitecturas se acercan, en cierta medida, al propósito de esta tesis, pero sus objetivos no son los mismos, y esto hace que no sean soluciones adecuadas debido a su relativa falta de flexibilidad, complejidad y alto grado de formalismo.

### 3.2.8. Metodologías de análisis y diseño

**OCL (Object Constraint Language)**

Respecto a OCL, también existe una coincidencia parcial de objetivos, y por eso se incluye en la revisión de tendencias del capítulo 2. Pero no constituye un precedente de resolución de esta tesis.

El propósito principal de OCL no pasa por la verificación automatizada que aquí se persigue. En [UML] se dice explícitamente que UML es un lenguaje para especificar, visualizar, construir y documentar los artefactos de los sistemas software, así como para modelado de negocios y otros sistemas no-software. OCL es una extensión de UML para especificar restricciones sobre los objetos de los modelos; es decir, su fin principal es la especificación. La verificación estática no es una motivación básica de OCL.

Por otra parte, no está garantizado que lo que forma parte de OCL sea ejecutable; es un lenguaje de modelado, no de programación, y como tal lenguaje de modelado está diseñado sobre todo para lectores humanos. Esto hace albergar dudas sobre el grado de automatización posible sobre estas expresiones.

Entre los propósitos de OCL mencionados en [Ra97] y presentados en 2.8.1, figura la especificación, pero no está diseñado de manera explícita para el uso que aquí se pretende. Por lo demás, tampoco tiene una semántica lo suficientemente abierta como para plasmar conocimiento de manera inmediata.

En conjunto, puede considerarse que OCL presenta (para el enfoque de esta tesis) algunos de los problemas de los métodos formales, y en cualquier caso se trata de un método de modelado y especificación y no de verificación de componentes.

**Catalysis**

No puede negarse el predicamento del que goza Catalysis por lo que se refiere al análisis y diseño orientado a componentes. Sin embargo, tampoco da una respuesta satisfactoria al problema aquí planteado.

Catalysis es un método de análisis, diseño, modelado y especificación. Al igual que ocurre con UML o con OCL, puede coincidir parcialmente con los objetivos de esta tesis, que pasan por evitar la ambigüedad en las descripciones del comportamiento de los componentes, pero no se centra de la misma manera en el problema de la verificación estática y automática.

El principal interés de Catalysis reside en resolver el problema de la construcción del software basado en componentes, desde la captura de requisitos hasta las diversas decisiones de diseño e implementación. Proporciona construcciones para dilucidar de forma inequívoca si el comportamiento de un elemento es o no conforme a lo esperado. Ofrece métodos para decidir la división en componentes o para crear estos componentes mediante procesos de generalización y abstracción. Permite separar la especificación de un sistema del diseño o implementación concretos de un caso específico, y verificar que las decisiones de diseño e implementación tomadas no contradicen las especificaciones (en otras palabras, que son *un refinamiento de estas*). Y ofrece apoyo, en fin, para otros muchos aspectos del modelado y desarrollo riguroso de software. Pero no tiene mucha relación con los objetivos planteados por nuestra tesis.

Si se consultan las características principales de Catalysis expuestas en 2.8.2, se verá que no hay referencias notables al proceso de verificación regresiva y automática que aquí se plantea. Como mucho, se menciona la precisión en las especificaciones y la importancia de la colaboración entre componentes, así como la existencia de conectores. Pero todo esto sigue siendo principalmente una herramienta "para uso humano"; por supuesto, no se cierra la puerta a posibles sistemas automáticos que comprueben la coherencia de un modelo, sistemas que también pueden construirse para OCL o para metodologías más antiguas. No obstante, en principio no se hace referencia directa a esta cuestión ni parece ser un factor de motivación de Catalysis.

Esto lo confirma una revisión de la lista de beneficios que según sus autores se derivan del uso de Catalysis (véase el apartado 2.8.2 de esta disertación o la pág. 40 de [SW99]), o de la lista alternativa de [CA00] (también en 2.8.2), que sería la percibida por los usuarios de Catalysis. Ninguno de ellos tiene relación directa con nuestra tesis.

Existen otros indicios significativos de esto. Recuérdese el ejemplo de aserciones redundantes mencionado en la página 64 de este documento; se aprecia claramente que esas aserciones van dirigidas a un lector humano. Los autores no se plantean que la restricción redundante pueda deducirse de las otras dos, porque proponen su inclusión por tener esta mayor poder expresivo de cara al lector.

En [SW99, pág. 313] se comenta la posibilidad de que aparezcan restricciones contradictorias en dos paquetes de especificaciones que se importan; esto no es motivo de preocupación. Simplemente, se ha modelado algo que no puede existir, con lo que tales contradicciones se detectarán cuando se intente implementar tal sistema. Obviamente, los autores no tienen en mente una detección automática de estas contradicciones.

Sobre la concepción que se tiene de los componentes, Catalysis propone una visión centrada en los beneficios ya mencionados en otros puntos de esta disertación o en [Sz97]: reutilización, mantenimiento, desarrollo en paralelo o por terceros, y otras (también incorpora las consabidas referencias a COM, CORBA o JavaBeans). En esta tesis se propone una visión del modelo de componentes como marco para la verificación del software, y tal aspecto del uso de componentes prácticamente no se menciona. Véase en

[SW99, pág. 397] una lista de posibles razones para pasarse al desarrollo basado en componentes; la verificación no es una de ellas, y está fuera de lugar hablar de verificación automatizada, asequible, basada en el conocimiento. Una de las pocas referencias a la validación que aparecen es la reflejada aquí en 2.8.2, donde se habla de "código de validación" al enumerar lo que incluye un paquete de componente, pero parece que se refiere a código ejecutable que realiza comprobaciones, y no a una validación estática similar a la planteada aquí.

Por último, Catalysis ofrece una manera de especificar las colaboraciones a través de los denominados conectores, pero tampoco tiene en cuenta el problema del emparejamiento de componentes de cara a una verificación como la exigida en esta tesis. El único patrón de actuación que tiene relación con la adecuación entre puntos de conexión es *Plug Conformance* (patrón número 10.4, pág. 449 de [SW99]). Allí se alude a que las especificaciones entre servicios ofrecidos y requeridos por los componentes pueden no encajar entre sí. Pero su planteamiento es similar al de cómo afrontar el problema de la documentación contradictoria; el diseñador detectará por sí mismo las incoherencias, confrontará las dos especificaciones e intentará conseguir un funcionamiento correcto solucionando los problemas planteados. El enfoque es muy diferente del nuestro.

Como conclusión, Catalysis es una metodología de gran influencia en el desarrollo basado en componentes, pero su fin principal es el modelado y el camino desde este a la implementación, y no la verificación estática, automática y asequible de la combinación de componentes.

### 3.2.9. Plataformas de componentes

La presentación del estado del arte por lo que se refiere a componentes tenía que incluir forzosamente una breve reseña de las aquí denominadas plataformas de componentes, y así se han presentado algunas de las iniciativas más representativas al respecto (capítulo 2.9). Como ya se razonó entonces, la comunidad del desarrollo de software identifica mayoritariamente la tecnología de componentes con estas plataformas.

No puede negarse que estas propuestas cuentan con la ventaja de una amplia difusión y disponibilidad, y que se utilizan frecuentemente en proyectos reales y en sistemas en producción. Esto no es óbice, sin embargo, para que no constituyan una respuesta satisfactoria a los objetivos planteados en esta tesis. De hecho, nuestro trabajo real con una de estas plataformas (COM) fue un factor de motivación importante para la misma.

Clemens Szyperski [Sz97, p. 171] denomina conjuntamente a estas plataformas **estándares de cableado**. Presenta así la razón de ser de estos estándares:

> *Hasta hace poco, la interoperabilidad del software estaba limitada a convenciones de llamada binarias a nivel procedimental. Todos los sistemas operativos definen convenciones de llamada, y todas las implementaciones de lenguajes respetan las convenciones de llamada de sus plataformas. Sorprendentemente, sin embargo, ninguno de los sistemas operativos tradicionales soportaba llamadas procedimentales que cruzasen los límites entre procesos. Incluso las llamadas de sistema a los servicios internos de un sistema operativo seguían con frecuencia convenciones de llamada no estándar. [...]*

Es decir, el planteamiento de origen de los estándares de cableado para componentes software se basa en las dificultades técnicas para realizar llamadas entre objetos o procedimientos construidos con diferentes lenguajes de programación o utilizados en diferentes sistemas operativos. Si se revisan las definiciones y declaraciones de propósitos que ofrece la documentación de cada uno de estos estándares, se advierte claramente este propósito de fondo.

Existen varias definiciones de *middleware*; aunque con ligeras diferencias, transmiten la misma idea. En el ámbito de las redes de ordenadores, una posible definición es:

**Def. XV**: Middleware es cualquier cosa de una red que funciona (en un nivel de un modelo de referencia de redes) por encima del nivel de transporte (TCP/IP) y por debajo del de aplicación (API) [Ai00].

Otra definición más genérica es [It01]:

**Def. XVI**: Middleware es software que hace de interfaz entre un cliente y un servidor.

Rosen [Ro00] ofrece una definición aún más abierta y hasta cierto punto tautológica:

**Def. XVII**: Middleware es "el software que está en el medio".

De alguna manera, las plataformas de componentes aquí evaluadas podrían calificarse genéricamente como middleware, y así se definen a sí mismas en ocasiones. Pero el middleware, por sí solo, no puede satisfacer las restricciones que en esta tesis se plantean.

Retomando las palabras de Szyperski [Sz97, p. 23]:

> *Los estándares de cableado no son suficientes. [...] Es tentador concentrarse en niveles que están libres de los problemas semánticos de tales áreas de más alto nivel [se refiere a aspectos específicos de la aplicación y/o el dominio]. Son buenos candidatos los estándares a nivel de "cableado" o "fontanería". Es obvio que los componentes necesitan conectarse entre sí para ser útiles. También es obvio que tales conexiones deben seguir estándares para hacer siquiera probable que dos componentes cualesquiera tengan "conectores" compatibles.*
>
> *[...] Cuando se adopta un enfoque que empieza con estándares y avanza hasta los productos, es natural que los primeros estándares en llegar afronten el nivel de "cableado". Si no existe nada real sobre lo que se pueda construir, es difícil embarcarse en esfuerzos de estandarización específicos del dominio o de la aplicación. Así, OMG publicó CORBA primero [...]. CORBA es un estándar de cableado que posibilita la comunicación entre objetos que han sido programados en diferentes lenguajes y soportados por diferentes sistemas operativos.*

Efectivamente, tanto COM como CORBA como JavaBeans son diferentes alternativas para poder integrar componentes. A un nivel básico se trata de puros "estándares de cableado", sobre los que se han construido diferentes conjuntos de servicios o marcos de referencia. Pero puede verse que ni en las definiciones de los interesados ni en las disquisiciones más generales acerca del middleware aparece referencia alguna a la *verificación* de la composición de componentes, si exceptuamos la lógica preocupación por la coherencia en lo que se refiere a convenciones de llamada o número y tipo de parámetros. Hay varios síntomas evidentes:

- Tanto COM como CORBA se basan en el uso de IDL o alguna variante del mismo para definir los interfaces. IDL no ofrece ningún recurso expresivo para describir

restricciones como las que se desea verificar en esta tesis; ofrece una simple variante de la verificación estricta de tipos dentro de un lenguaje de programación.

- JavaBeans ofrece el mecanismo de introspección, pero este mecanismo no permite especificar requisitos ni garantías funcionales [Su97]. Está orientado a describir interfaces en el sentido clásico: tipos y nombres de clase, propiedades, eventos, métodos y parámetros. En lo básico, no deja de ser una versión dinámica (en tiempo de ejecución) de la información que proporciona el IDL.
- Ninguna de estas plataformas proporciona, de por sí, mecanismos adicionales de verificación para cuando se ensamblan componentes.
- El crecimiento posterior de estos modelos ha sido hacia un middleware más potente en lo que se refiere a interacción entre máquinas o adaptación entre diversos estándares, o bien a estandarizar interfaces y construcciones de más alto nivel, pero en ningún caso se plantea ampliar las capacidades de verificación de las conexiones. Se busca encontrar nuevas posibilidades de conectar e integrar objetos, no detectar problemas de integración que pueden darse con los mecanismos ya disponibles.

Por tanto, básicamente las plataformas de componentes mencionadas no ofrecen mecanismo alguno para aprovechar todo el conocimiento disponible sobre un componente (que evidentemente va mucho más allá de sus métodos y parámetros) en un proceso automático y estático de verificación. La subsanación de esta deficiencia es, precisamente, el objeto de estudio de esta tesis.

## 3.3. Interés del problema

En primer lugar, el interés de verificar la combinación de componentes responde a la creciente tendencia a construir sistemas software basados en componentes, y a la importancia de detectar los defectos del software; teniendo en cuenta que muchos defectos no tienen su origen en un componente individual, sino en la combinación de varios componentes, que resulta ser inadecuada, parece oportuno desarrollar técnicas encaminadas a detectar ese tipo de defectos (técnicas que serían complementarias de otras, como por ejemplo las pruebas o las revisiones técnicas formales [NASA93]).

Esta tesis se centra en la detección de problemas que surgen al *combinar* componentes (entendiendo por *combinar* el interconectar componentes para formar un sistema); de alguna manera, se pretende verificar que las uniones *entre componentes* son correctas. ¿Por qué en esta tesis se da por hecho que cada componente, internamente, es correcto? Por lo que se refiere al software, en general es realmente difícil asegurar que un componente no presenta fallos internos, pero aquí se asume por dos razones:

1. Delimitar el campo de actuación de la tesis, dejando fuera problemas que, pudiendo ser interesantes, no constituyen un objetivo de la misma. El eje de esta tesis es la detección de problemas que se originan al *combinar* componentes y no al *construirlos*.
2. Aunque se renuncia a estudiar el interior de un componente como algo específico, se cree posible aplicar las mismas ideas de manera recursiva; si un componente deja de ser una caja negra y se considera a su vez un subsistema formado por componentes, también el interior de un componente podrá verificarse (hasta cierto punto) con los mismos métodos, y esto será aplicable hasta llegar a un nivel

irreducible de componentes axiomáticamente correctos (quizás las bibliotecas o bases sobre las que se construye el proyecto actual, quizás los propios operadores del lenguaje de programación...)

Por tanto, el hecho de verificar la corrección *inter-componente* sin entrar a considerar la corrección *intra-componente* puede parecer una limitación, pero en realidad es una estrategia deliberada: proporcionará un modelo más simple, que sin embargo puede ser aplicado una y otra vez de menor a mayor nivel de detalle, por el procedimiento de considerar cada nuevo elemento individual como un sistema cuando llegue el momento de examinar su interior.

De hecho, en los programas (incluyendo los que no se han construido mediante componentes según la acepción más extendida) se producen muchos tipos de fallos que son susceptibles de ser expresados en términos de deficiente combinación de componentes. En el desarrollo modular u orientado a objetos, esta metáfora puede ser aplicada en multitud de ocasiones. Yendo más lejos, incluso una simple división por cero puede considerarse como una violación de las condiciones de funcionamiento de un componente –el operador de división- al colocar este en un entorno en conexión con otros componentes y suministrarle una entrada errónea –el denominador cero-. Por tanto, la metáfora del sistema formado por componentes interconectados puede ser útil para prevenir muy diversos tipos de errores del software, y el interés de esta tesis iría más allá de la corriente principal de los componentes software (véase Def. I en página 8).

Por lo que se refiere a la verificación **estática**, la motivación de la idea es que la prueba (dinámica) de sistemas plantea importantes dificultades. Como ya se ha dicho, el espacio de estados correspondiente a un sistema de software es, excepto en casos triviales, muy extenso, y un recorrido exhaustivo de verificación no suele ser viable. Este problema se suele afrontar mediante la preparación de casos de prueba selectivos y significativos, guiada por ciertos heurísticos, tales como la comprobación de valores límite; pero en general esto no garantiza la corrección del sistema. Muchas veces, gracias al conocimiento que se tiene de un sistema, se puede predecir que se producirá un problema (bastaría con percatarse a tiempo del detalle adecuado), mientras que detectar ese problema mediante pruebas puede resultar mucho más difícil. Un método estático que aprovechase el conocimiento apriorístico aportaría por tanto elementos positivos de cara a la detección de defectos, descargando responsabilidad de las pruebas.

En cualquier caso, la estrategia de pruebas implica tener implementado el sistema, al menos hasta cierto punto. Una verificación estática permitiría detectar los problemas en una etapa más temprana del ciclo de desarrollo, en tiempo de diseño. Es norma ampliamente aceptada que, en el proceso de desarrollo del software, cuanto antes se detecte un problema más fácil y barato resultará solucionarlo [Pr92]; el sistema aquí propuesto sería ventajoso porque no se resignaría a esperar para detectar los problemas, sino que intentaría hacerlo *mientras el sistema se construye*.

Un tercer motivo para pensar en verificación estática es que las pruebas detectan *las consecuencias o efectos* de un defecto, no el defecto en sí mismo; tras detectar las consecuencia de un error, normalmente hay que realizar un trabajo de diagnóstico y depuración que permita señalar cuál es el defecto que causa el comportamiento indeseado, tarea que a veces es trivial pero otras veces resulta extraordinariamente costosa. Por contra, la verificación estática en general realiza algún tipo de análisis que, sin ejecutar el programa, permite señalar defectos en el mismo; por tanto, no se están señalando consecuencias, sino los propios defectos y su porqué.

Esto entronca también la expresión **con el conocimiento disponible** que forma parte del enunciado. En contraste con las dificultades que plantean las pruebas, los desarrolladores de un componente tienen con frecuencia un gran conocimiento sobre el propósito del mismo y sobre sus condiciones correctas de funcionamiento. Este conocimiento no siempre resulta fácil de expresar en forma de pruebas, y es aún más difícil transmitírselo al usuario del componente con la garantía de que este va a tener presentes (y respetar) todas las restricciones aplicables. Por tanto, a efectos prácticos ese conocimiento se pierde. La definición de las interfaces de los componentes es, habitualmente, muy pobre; es habitual que las únicas verificaciones que se realizan de forma consistente y automatizada sean verificaciones de tipos / signaturas, y el resto de directrices queden expresadas en documentación escrita en lenguaje natural. Pasa a ser responsabilidad del desarrollador tener presentes esas directrices y verificarlas, cosa que resulta verdaderamente difícil en cualquier sistema de tamaño respetable.

Lo más frecuente es que los componentes se caractericen simplemente por las signaturas de las interfaces que ofrecen. El incorporar todo el conocimiento posible a los componentes permite realizar verificaciones adicionales.

La verificación **automática** persigue precisamente marcar una diferencia con la situación actual, en la que las especificaciones de un componente (signaturas aparte) se realizan con frecuencia en lenguaje natural y la observancia de estas especificaciones resulta difícil; en primer lugar porque estas especificaciones suelen ser ambiguas o insuficientes, y en segundo lugar porque en cualquier caso esta es una tarea en la que los humanos cometen, lógicamente, errores y omisiones. Además, el que el proceso sea automático fomenta la verificación *regresiva*, de modo que cuando se modifica un sistema sea fácil comprobar si tal modificación rompe sus reglas de funcionamiento, ya que la verificación es repetible a un coste relativamente bajo. El que la verificación sea automática obliga, además, a que sea un proceso relativamente formal y no ambiguo, como ocurre frecuentemente con las especificaciones realizadas y verificadas en lenguaje natural.

Por lo que respecta a la verificación **asequible**, y todo lo que aquí se ha englobado en ese término (véase el apartado1.1), lo que se persigue es hacer viable la integración de esta práctica en el modelo de desarrollo. Otros métodos de verificación, por diversas razones (requerir un alto grado de especialización y/o formación, estar basados en técnicas pioneras y poco difundidas, ser difíciles de implementar en la práctica, tener un campo o ámbito de aplicación muy específico), no han conseguido una amplia aceptación.

Se pretende un sistema susceptible de ser implementado mediante *técnicas conocidas y viables* porque esto disminuye costes –y riesgos- de implementación. Las empresas de desarrollo de software tienen unos recursos limitados para una tarea que ya de por sí es costosa, compleja y arriesgada. Si el sistema de verificación plantea alguno de los siguientes impedimentos, resultará difícil que las organizaciones adopten este método:

- Basarse en técnicas complejas y poco probadas (propensas a errores).
- Requerir un avance significativo del estado del arte en diversas áreas.
- Exigir el uso de dispositivos software de gran envergadura desarrollados de forma específica.

Se pretende que el sistema sea *susceptible de ser comprendido y aplicado en la práctica con relativa facilidad* por personal de desarrollo por varias razones. En primer lugar porque disminuye

costes de formación. En segundo lugar porque requiere un grado de especialización menor y plantea menos restricciones de selección de personal. Y en tercer lugar porque el personal será mucho más productivo (y aplicará el método con mayor frecuencia y aprovechamiento) si es capaz de dominarlo. Por tanto parece razonable que el sistema sea lo más simple posible.

Se pretende, al fin, que el sistema sea *flexible y susceptible de ser aplicado en diferentes ámbitos* (no demasiado específico) porque esto mejora el retorno de inversión y por tanto también conlleva una disminución del coste de adopción de la técnica. Algunos métodos de verificación van dirigidos a problemáticas muy concretas; las resuelven satisfactoriamente, pero no proporcionan la versatilidad que permita aprovecharlos para otros usos. La simplicidad mencionada en el párrafo anterior también va en beneficio de la flexibilidad.

Como último punto en la justificación del interés de esta tesis, hasta donde llega nuestro conocimiento (y tras la pertinente evaluación del estado del arte) no tenemos noticia de un sistema de verificación basado en estas ideas y que atienda estos condicionantes, es decir, no conocemos que se haya dado previamente una respuesta satisfactoria a la cuestión que se plantea.

## 3.4. Resultados alcanzados

En el proceso de desarrollo de esta investigación se han alcanzado diversos resultados que desde nuestro punto de vista apoyan satisfactoriamente nuestra tesis.

Inicialmente, y tras la evaluación de las tendencias vigentes en el terreno de los componentes software que tuvieran relación con nuestra tesis, el trabajo planteado era idear un método que diese respuesta a la misma. Para ello, parecía necesario ir probando en la práctica las diferentes ideas que fuesen surgiendo. Esto ayudaría en primer lugar a perfilar dichas ideas y concretarlas, pero también ayudaría a sopesar en todo momento la viabilidad del método, en una doble vertiente: la posibilidad real de implementar herramientas que le diesen soporte, y la posibilidad de aplicar este método a diversos problemas prácticos.

Por tanto, el desarrollo de la investigación tomó la forma de un ciclo iterativo, en el que se formulaba parte del modelo, se implementaba un prototipo de herramienta que le diese soporte, y se aplicaba dicha herramienta a algún problema de ejemplo. Esto permitía obtener enseñanzas para refinar el modelo y para mejorar el prototipo.

### 3.4.1. Prototipos y facilidad de implementación

Como primer resultado, cabe citar el desarrollo de los prototipos mencionados (que se describen en detalle en el capítulo 5). Estos prototipos se desarrollaron con recursos humanos muy limitados (una sola persona a tiempo parcial) y también con recursos técnicos claramente accesibles (tecnologías convencionales de desarrollo).

Como resultado de su uso, se puede calificar como muy viable la técnica propuesta desde el punto de vista de las herramientas. Aunque el uso de estos programas no resulta excesivamente *amigable*, ese no era su propósito, y los problemas de usabilidad mencionados parecen fácilmente resolubles con un moderado esfuerzo de desarrollo. El resultado relevante es que el modelo se comportó exactamente como estaba previsto desde el punto de vista técnico: se pueden implementar herramientas que se basan en Itacio, con tecnologías convencionales y ampliamente disponibles.

### 3.4.2. Facilidad de uso

Itacio se basa, en su formulación actual, en lógica de primer orden bajo la forma de cláusulas Horn. Se trata de una técnica relativamente simple y asequible, ampliamente conocida. Para verificar esta afirmación, basta con examinar planes de estudio o modelos curriculares universitarios. Un estudio de este tipo puede verse en [Lb99]. Evaluando diversos modelos curriculares, se aprecia una constante presencia de los fundamentos de la Lógica y la Programación Lógica. En el currículo de Carnegie-Mellon [Sa85] aparece en el primer nivel un curso Matemáticas Discretas (Ref. 150), en el que se imparten temas de habilidades lógicas, inducción, introducción a la lógica y al razonamiento matemático y álgebra de Boole. En el tercer nivel se imparte Lógica para Informática (Ref. 351). En el currículo de ACM/IEEE-91 (que prácticamente es la expresión institucional de la profesión informática a nivel mundial) aparece también como imprescindible para un titulado en informática haber cubierto los temas de matemáticas discretas, lógica básica de predicados y de proposiciones, álgebra de Boole, técnicas de prueba (incluyendo inducción y contradicción), etc. Como opción se proponen también asignaturas que profundicen en la lógica matemática. El Programa Modular UNESCO-IFIP 94 incide en los mismos temas: el módulo 3.1, Lógica para Informática, incluye sintaxis y semántica de lenguajes lógicos de primer orden, bases lógicas de la programación lógica (con mención expresa a las fórmulas Horn), unificación y cálculo de resolución, etc. Otros módulos de este currículo completan estas enseñanzas (como el módulo 4.2, Introducción a la Inteligencia Artificial). Las diversas universidades extranjeras refrendan esta presencia de los fundamentos de la programación lógica en sus planes de estudios.

Evaluando las universidades españolas, se observa una tónica similar. Los fundamentos teóricos de lógica se imparten en la inmensa mayoría de los planes de estudios para titulaciones informáticas. En la Tabla 3 (fuente: [Lb99]) se observa una clasificación de algunas universidades españolas según el nivel al que se imparte Lógica.

La Tabla 4, por su parte, muestra el resultado de un estudio informal realizado específicamente para esta tesis sobre los planes de estudios de 25 universidades españolas elegidas al azar. De las 25 universidades evaluadas, 22 impartían titulaciones oficiales de Informática (ya fuesen estudios medios o superiores). En todas ellas se impartían, en mayor o menor grado, asignaturas relacionadas con las técnicas de inferencia utilizadas en Itacio (asignaturas de lógica, inferencia, inteligencia artificial, ingeniería del conocimiento, programación lógica, etc.). Las asignaturas concretas y los cursos en los que se inscriben figuran en la tabla mencionada, en la que, en la columna "Curso", se han utilizado las abreviaturas siguientes para las titulaciones:

- ITIG – Ingeniería Técnica en Informática de Gestión
- ITIS – Ingeniería Técnica en Informática de Sistemas
- II – Ingeniería en Informática

| Nivel | Universidad |
|---|---|
| Introductorio | Universidad de Zaragoza<br>Universidad de La Laguna<br>Universidad Carlos III de Madrid<br>Universidad Politécnica de Valencia |
| Medio | Universidad Autónoma de Barcelona<br>Universidad Complutense de Madrid<br>Universidad de Alicante<br>Universidad de Castilla-La Mancha<br>Universidad Politécnica de Cataluña<br>Universidad de Lérida<br>Universidad de Murcia<br>Universidad de las Islas Baleares |
| Alto | Universidad Politécnica de Madrid<br>Universidad de Málaga |

*Tabla 3. Niveles a los que se imparte Lógica en algunas Universidades Españolas.*

| Universidad | Curso | Asignatura |
|---|---|---|
| Mondragón | 3º ITIS | Lógica formal |
| | 5º II | Sistemas expertos y redes neuronales |
| Alfonso X El Sabio | 2º II | Lógica y Teoría de Autómatas (6cr.) |
| Antonio de Nebrija | 1º ITIS | Lógica formal (complementaria, 6cr.) |
| | 1º II | Lógica formal (básica y obligatoria, 6cr.) |
| Autónoma de Madrid | 3º II | Inteligencia artificial (obligatoria, 7,5 cr.) |
| Camilo José Cela | 4º II | Inteligencia artificial e Ingeniería del Conocimiento (obligatoria, 9 cr.) |
| Carlos III de Madrid | 1º II | Lógica computacional (obligatoria, 6cr.) |
| Complutense de Madrid | 1º ITIG/S | Lógica (obligatoria, 4,5 cr.) |
| | 1º II | Lógica (obligatoria, 4,5 cr.) |
| | 3º II | Programación funcional (obligatoria, 4,5 cr.)<br>Programación lógica (obligatoria, 4,5 cr.) |
| | 4º II | Inteligencia artificial e Ingeniería del conocimiento (troncal, 9 cr.) |
| | 5º II | Programación declarativa avanzada (optativa, 6 cr.) |
| Alcalá de Henares | 1º II | Lógica para la computación (obligatoria, 4,5 cr.) |
| | 4º II | Inteligencia artificial e Ingeniería del conocimiento (troncal, 9 cr.) |
| | ITIS | Inteligencia artificial (optativa, 4,5 cr.) |
| | ITIG | Inteligencia artificial (optativa, 4,5 cr.) |
| Alicante | 1º ITIG/S | Lógica de primer orden (obligatoria, 4,5 cr.) |
| | ITIG/S | Fundamentos de Inteligencia artificial (optativa, 4,5 cr.) |
| | 1º II | Lógica de primer orden (obligatoria, 4,5 cr.) |
| | 4º II | Fundamentos de Inteligencia artificial (troncal, 4,5 cr.)<br>Técnicas de Inteligencia artificial (troncal, 4,5 cr.)? (sin datos) |
| Almería | ITIG/S | Fundamentos de la Inteligencia Artificial y Sistemas Expertos (optativa, 7,5 cr.) |
| | 1º II | Inteligencia Artificial e Ingeniería del Conocimiento (troncal, 9 cr.) |
| | II | Ingeniería de Sistemas Basados en Conocimiento (optativa, 4,5 cr.)<br>Programación Lógica y Funcional (optativa, 6 cr.) |
| Burgos | ITIG | Sistemas Expertos e Inteligencia Artificial (optativa, 9 cr.) |
| | 5º II | Inteligencia artificial e Ingeniería del conocimiento (troncal, 9 cr.) |
| | II | Gestión del conocimiento (optativa, 6 cr.) |

| Universidad | Curso | Asignatura |
|---|---|---|
| Cantabria | No imparte estudios oficiales de Informática | |
| Castilla la Mancha | 1º ITIG/S (Ciudad Real y Albacete) | Lógica (obligatoria, 6 cr.) |
| | 1º II (Ciudad Real y Albacete) | Lógica (obligatoria, 6 cr.) |
| | 2º II (Ciudad Real y Albacete) | Programación declarativa (obligatoria, 9 cr.) |
| | 4º II (Ciudad Real y Albacete) | Inteligencia artificial e Ingeniería del conocimiento (troncal, 9 cr.) |
| | II (Ciudad Real) | Modelos y aplicaciones de Inteligencia Artificial (optativa, 4,5 cr.) |
| | 4º II (Albacete) | Ingeniería del conocimiento (obligatoria, 9 cr.) |
| | 5º II (Albacete) | Ampliación de programación declarativa (optativa, 4,5 cr.)<br>Verificación automática? (optativa, 4,5 cr.)<br>Resolución de problemas de Inteligencia Artificial (optativa, 4,5 cr.) |
| Cádiz | 2º ITIG | Programación declarativa (optativa, 6 cr.) |
| Córdoba | ITIG/S | Lenguajes de Inteligencia Artificial (optativa, 4,5 cr.) |
| Deusto | 4º II | Inteligencia artificial e Ingeniería del conocimiento (troncal, 9 cr.) |
| | 5º II | Gestión del conocimiento (optativa, 6 cr.) |
| Extremadura | 4º II | Inteligencia artificial e Ingeniería del conocimiento (troncal, 9 cr.) |
| | II | Lógica y computabilidad (optativa, 6 cr.) |
| Granada | ITIG/S | Modelos de la Inteligencia Artificial (optativa, 6 cr.) |
| | 1er ciclo II | Ingeniería del conocimiento (obligatoria, 7,5 cr.) |
| | 2º ciclo II | Inteligencia artificial e Ingeniería del conocimiento (troncal, 9 cr.)<br>Modelos de la Inteligencia Artificial (obligatoria, 4,5 cr.) |
| | II | Ingeniería del conocimiento: ampliación (optativa, 6 cr.)<br>Lógica informática (optativa, 6 cr.) |
| Huelva | 3º ITIG/S | Sistemas expertos (optativa, 6,75 cr.)<br>Aplicaciones de la Inteligencia Artificial (optativa, 4,5 cr.)<br>Lógicas clásicas y no clásicas (optativa, 4,5 cr.) |
| Jaén | ITIG | Programación declarativa (optativa, 7,5 cr.)<br>Introducción a la Inteligencia Artificial (optativa, 7,5 cr.) |
| La Rioja | No imparte estudios oficiales de Informática | |
| Islas Baleares | 1º ITIG/S | Lógica (obligatoria, 4,5 cr.) |
| | 4º II | Inteligencia Artificial (troncal, 9 cr.) |
| Murcia | 2º ITIG/S | Lógica Computacional (optativa, 6 cr.) |
| | 3º ITIG/S | Introducción a la Inteligencia Artificial (optativa, 6 cr.) |
| | 1º II | Lógica Computacional (obligatoria, 4,5 cr.) |
| | 2º II | Técnicas de Inferencia (obligatoria, 4,5 cr.) |
| | 4º II | Inteligencia Artificial (troncal, 6 cr.)<br>Modelos de Inteligencia Artificial (4,5 cr.) |
| | 5º II | Ingeniería del conocimiento (obligatoria, 6 cr.) |
| | 4º/5º II | Adquisición de Conocimiento (optativa, 4,5 cr.) |
| Navarra | No imparte estudios oficiales de Informática | |
| Oviedo | 1º ITIG/S (Gijón) | Lógica (obligatoria, 4,5 cr.) |

| Universidad | Curso | Asignatura |
|---|---|---|
| | ITIG ( Gijón) | Introducción a la Inteligencia Artificial (optativa, 6 cr.)<br>Minería de Datos y Extracción de Conocimiento (optativa, 6 cr.)<br>Programación Declarativa (optativa, 6 cr.) |
| | 1º ITIG/S (Oviedo) | Lógica (obligatoria, 6 cr.) |
| | ITIG/S (Oviedo) | Programación Lógica y funcional (optativa, 6 cr.) |
| | 4º II | Inteligencia Artificial (obligatoria, 9 cr.) |
| | 5º II | Ingeniería del Conocimiento (obligatoria, 9 cr.) |

*Tabla 4. Presencia de la lógica, programación lógica y asignaturas relacionadas en los planes de estudios de algunas universidades españolas.*

La difusión de que gozan las técnicas de lógica y programación lógica entre los profesionales no es el único argumento que apoya la facilidad de uso; es otro hecho importante que hay disponibles muchas herramientas de inferencia basadas en lógica de primer orden, sobre todo en su implementación en el lenguaje Prolog y otros similares. Estas herramientas cuentan con un largo historial dentro de la Informática.

El que Itacio base su mecanismo de representación en técnicas asequibles y conocidas redunda en una gran facilidad de uso. El usuario no necesita un cambio de mentalidad ni una formación específica en métodos formales de gran envergadura; la representación de restricciones en Itacio se lleva a cabo mediante un simple lenguaje de programación, declarativa en este caso. Además, la lógica está abierta a la representación de muy diversos conocimientos; basta con declarar los predicados oportunos, cuya semántica aporta el propio usuario. Esto se ha podido comprobar en los experimentos realizados.

Consideramos que el haber llegado a un sistema de verificación y validación basado en herramientas conceptuales fáciles de asimilar es uno de los resultados que diferencian esta propuesta de otras.

### 3.4.3. Simplicidad

Enlazando con la facilidad de uso que se consigue al apoyar el sistema formal de Itacio en la lógica de primer orden en detrimento de otras posibilidades (tales como alguno de los muchos métodos formales que existen, las álgebras de procesos, etc.) está el tema de la simplicidad del modelo.

El modelo Itacio es extremadamente simple. Esto se ha perseguido de forma deliberada, de acuerdo con la motivación presentada para esta tesis. Incluye pocos conceptos y el algoritmo de generación de la base de conocimientos es también relativamente simple. Se han introducido los elementos mínimos para que el sistema fuese operativo, y se ha hecho un esfuerzo consciente por no incluir "adornos" que pudiesen complicarlo.

Esta simplicidad puede también considerarse un logro, ya que la mayoría de los modelos comparables resultan notablemente más complejos. Por supuesto, tanto la simplicidad como la complejidad tienen sus ventajas y sus inconvenientes; un modelo más complejo será, seguramente, más específico, y contará con un mayor grado de adaptación a ciertos problemas. También ofrecerá más recursos listos para ser utilizados, resolviendo necesidades concretas. Un modelo más simple ofrece sólo una base, que es necesario ampliar con definiciones propias si se quiere hacer frente a problemas más específicos. Pero ciertamente, esta tesis exigía un modelo simple por razones que se explican en otros puntos

de esta disertación; Itacio no representa un intento de desplazar a otros métodos formales allí donde estos son fuertes, sino aportar soluciones en las áreas en las que no resultan del todo adecuados. El haber sintetizado un modelo simple que cumple los objetivos señalados es, pues, un resultado fundamental en esta tesis.

### 3.4.4. Flexibilidad

Un resultado importante es también el que Itacio pueda aplicarse a problemas muy diversos. En el capítulo 5 se describe su uso para afrontar diversos problemas, la mayor parte de los cuales se han ido eligiendo de forma "improvisada" en el sentido de que ni el modelo ni las herramientas se diseñaron pensando en ellos. Sin embargo, tanto el modelo como las herramientas (prototipos, en este caso) pudieron utilizarse sin necesidad de ninguna modificación "hecha a medida". Esta es una propiedad fundamental del modelo.

### 3.4.5. Niveles de abstracción

Además, puede pensarse que la simplicidad del modelo exige un gran esfuerzo de desarrollo para poder aplicarlo. En un apartado anterior se mencionaba que la generalidad del modelo puede aparejar la necesidad de "rellenar" la distancia semántica entre este y el problema concreto, lo que exigiría muchas definiciones de base.

En realidad, no ha sido así. En algunos casos concretos esto puede ser cierto (y está previsto en el modelo a través de las llamadas *bibliotecas*); por ejemplo, la verificación de contratos de reutilización (véase capítulo 5.2.2) requiere cierta infraestructura de cara al proceso de inferencia (las reglas sobre contratos *bien formados*). Pero esto se debe a que el dominio del problema es bastante particular y requiere "cálculos internos". Para muchos otros usos, la aplicación de Itacio es prácticamente inmediata.

Esto es así gracias a que la semántica se incorpora a través de predicados de lógica de primer orden, y el usuario construye y define estos predicados según sus preferencias. En consecuencia, los predicados pueden ubicarse a cualquier nivel de abstracción. Esta flexibilidad de la lógica de primer orden ha quedado patente en diversas aplicaciones. Por ejemplo, al aplicar Itacio a un sistema de procesamiento de sonido en tiempo real (capítulo 5.2.4) los predicados utilizados constituyen casi una expresión literal del vocabulario del dominio del problema. Esto hace que la supuesta distancia semántica entre el modelo Itacio (muy general) y el problema concreto quede mitigada en gran medida. Paradójicamente, un modelo muy general como este es fácil de aplicar a diversos problemas específicos sin gran esfuerzo, gracias a la versatilidad de la lógica.

### 3.4.6. Limitaciones del modelo

Dentro de los resultados alcanzados, conviene mencionar también algunas limitaciones de este enfoque. En general, estas limitaciones estaban previstas en la concepción de este modelo, y el desarrollo de la tesis ha servido para corroborar las impresiones iniciales.

**Nomenclatura inconsistente**. El primer problema que cabe mencionar es el que surge, sobre todo, al integrar componentes de fuentes diversas: la nomenclatura inconsistente. Puede darse el caso de que en las expresiones restrictivas de dos componentes se utilicen los mismos nombres de predicado para representar hechos o reglas que son distintos, que hacen referencia a distinta información. También puede darse el caso opuesto: que se haga

referencia a la misma información bajo nombres de predicado distintos. Puesto que la lógica de primer orden permite que la semántica (desde el punto de vista humano) de las reglas quede implícita en los nombres de los predicados y átomos, los problemas de nomenclatura son a efectos prácticos problemas de conocimiento expresado de forma inconsistente, lo que dificulta o imposibilita los procesos de inferencia.

Este problema no es exclusivo de Itacio. La consistencia o estandarización de la nomenclatura afecta también a la composición de interfaces en sentido clásico, a las llamadas a cualquier API, a la compilación conjunta de módulos de código fuente, al enlazado (*linking*) durante el proceso de compilación, etc. Siempre que se unen entidades de software debe existir un acuerdo previo sobre la nomenclatura de los elementos implicados. Este problema se agudiza si se están integrando elementos de distintos fabricantes, como frecuentemente sucede en el campo de los componentes software.

En Catalysis, por ejemplo, puede darse este caso al combinar diversos paquetes (*packages*), lo que hace que el problema pueda afectar a muy diversos artefactos software (los paquetes de Catalysis pueden incluir código fuente, código compilado, nombres de variables, requisitos, documentación "narrativa", diagramas...) La postura de esta metodología ante este problema es, simplemente, resolverlo mediante mero renombramiento ([SW99, pág. 314]).

La conclusión es que el problema de la consistencia de nombres no es privativo de Itacio; es omnipresente dentro de la informática, y lo único que se puede hacer al respecto es utilizar los diversos medios existentes a nivel general (establecer renombramientos y correspondencias entre términos, realizar un esfuerzo previo de estandarización, etc.) y aplicar las técnicas de gestión del conocimiento oportunas.

**Gestión del conocimiento**. Básicamente, Itacio permite incorporar conocimiento a las interfaces de los componentes. La expresión de este conocimiento resulta una tarea asequible, pero por supuesto no puede calificarse de trivial, y abre una puerta para muchas mejoras. Aunque Itacio ofrece un cauce adecuado para dar uso a ese conocimiento, una vez se tiene esa posibilidad habría que pensar en manejar un conocimiento *de calidad* desde diversos puntos de vista. Esto implicaría intentar que las expresiones restrictivas fuesen completas, mínimas, no redundantes, no contradictorias, que permitiesen un proceso de inferencia eficiente en términos de velocidad y memoria...

Siendo una cuestión importante, no se encuentra entre los objetivos iniciales de Itacio el realizar una gestión óptima del conocimiento, y por eso en el modelo no se incluyen consideraciones al respecto (al igual que se utiliza la Programación Lógica con Restricciones pero no se incluyen en esta disertación detalles sobre cómo implementar un motor de inferencia CLP). En este caso, las técnicas aplicables provendrán del campo de la Ingeniería del Conocimiento, un terreno de investigación con entidad propia y que excede claramente el ámbito de esta tesis.

**Incumplimiento de expresiones**. Como ya se ha señalado en otros puntos de esta disertación, el que las expresiones restrictivas sean algo relativamente independiente del componente (lo "acompañan", pero no necesariamente tienen un vínculo directo con su implementación) hace que puedan no describir con exactitud su comportamiento. Es decir, una expresión restrictiva, en especial una garantía, puede *mentir*.

Esto es algo conocido desde el principio, y aceptado de manera consciente. Se entiende que lo importante es tener una vía para plasmar y utilizar el conocimiento sobre los componentes. Un componente puede tener algún defecto que hace que no se comporte

como se creía (es decir, como hacía pensar el *conocimiento* que se tenía sobre él). Ciertamente, Itacio no detectaría eso, y realizaría sus inferencias sobre suposiciones que en realidad son falsas; pero este problema simplemente se hereda, no se *agrava* en Itacio. Sin Itacio, se tiene un componente que no responde al comportamiento esperado, pero ni siquiera se comprueba si el comportamiento esperado plantea problemas al usar el componente. Con Itacio, en el caso peor, se tiene un componente que no responde al comportamiento esperado, pero al menos se detectan los problemas relacionados con ese comportamiento esperado.

Las alternativas que existen para detectar esos defectos y analizar el código ya han sido evaluadas y se han puntualizado sus puntos fuertes y débiles; en el caso de Itacio, la limitación que estamos describiendo, si bien existe, se ha aceptado de forma consciente para poder disfrutar a cambio de otras ventajas (por ejemplo, poder verificar un diseño sin que ni siquiera exista el código aún).

# 4. Descripción del modelo Itacio

En este capítulo se ofrece una descripción más formal del modelo Itacio. El objetivo de esta descripción es expresar las ideas planteadas previamente de manera menos ambigua y haciendo uso de una notación un tanto más precisa.

Se hará una presentación del modelo sin justificar los conceptos introducidos (excepto en algunos casos en los que, al tener las definiciones un carácter *constructivo*, resulta conveniente conocer la motivación de los diferentes pasos); posteriormente, se discutirá su propósito y su papel de cara a los objetivos que se persiguen en la tesis.

## 4.1. Por qué Itacio

> *Encerré en precioso mármol para la mansión eterna el tierno cuerpo de Itacio.*

Esa es la traducción de la inscripción que figura en la tapa del sarcófago de Itacio [Be84]. Se trata de un sarcófago visigodo de mármol blanco, que algunos autores suponen data del siglo V, y se conserva en el Panteón de los Reyes de la Catedral de Oviedo. Su decoración recoge influencias bizantinas, y parece ser que no se construyó en Asturias, sino que su origen se sitúa en Zamora y se cree que llegó a la región en tiempos de la monarquía (probablemente de Alfonso II) tras alguna incursión militar. No se sabe con certeza quién era Itacio (o Itacius en el original), aunque de la inscripción parece deducirse que falleció joven.

Se tiene, pues, una sólida envoltura creada para proteger algo delicado y misterioso. Una envoltura que lleva impresa, bien visible, su propia misión, dejando saber lo justo sobre su propósito y prácticamente nada sobre lo que guarda en su interior. Una envoltura, además, construida para perdurar eternamente y resistir el paso de los siglos. Un objeto, finalmente, que otros han fabricado pero que los asturianos hemos hecho nuestro, integrándolo con el resto de nuestros propios tesoros artísticos.

Además de hacer una pequeña concesión a la alegoría, a las raíces culturales y a la historia del arte, en vista del párrafo anterior y de lo que ya se ha explicado sobre los propósitos de esta disertación no parece mal nombre para un modelo de componentes.

# 4.2. Nociones básicas y definiciones

## 4.2.1. Componente

**Componentes**: Un *componente* C es una entidad que consta de una *frontera* F y un conjunto de *expresiones restrictivas* E.

Dado un componente C, se utilizará también la notación F(C) y E(C) para designar la frontera y el conjunto de expresiones restrictivas de ese componente. El conjunto de todos los posibles componentes se denota por **C**.

La **frontera** de un componente es un conjunto finito cuyos elementos se denominan *puntos de conexión*. Un punto de conexión puede ser una fuente ($s$) o un sumidero ($k$), con lo que en realidad F está dividida en dos subconjuntos disjuntos:

$$F = S \cup K$$

$$S \cap K = \varnothing$$

donde

$$S = \{s_1, s_2, \ldots, s_n\}$$

$$K = \{k_1, k_2, \ldots, k_m\}$$

El dominio de todos los puntos de conexión se denota por **A** (de "átomos").

Se hará uso de la notación S(C) y K(C) para designar, respectivamente, al conjunto de fuentes y el conjunto de sumideros de un componente dado C. Además, nótese que cuando se vean involucrados varios componentes se hará uso de índices para referirse a dichos componentes o a sus partes. También puede utilizarse la *notación de punto* para calificar los puntos de conexión. Los nombres calificados incluyen el nombre de componente para evitar ambigüedades, de modo que, por ejemplo, la fuente $s_i \in S(C_n)$ pasa a ser $C_n.s_i$.

## 4.2.2. Expresiones restrictivas

Las expresiones restrictivas también se dividen en dos conjuntos disjuntos: el conjunto de *requisitos*, R, y el conjunto de *garantías*, G. Ambos conjuntos están formados por predicados de lógica de primer orden sobre los puntos de conexión del componente, bajo la forma de cláusulas Horn [Sm95, LF98, EK76]. El dominio de los predicados se denota por **P**.

$$E = R \cup G$$

$$R \cap G = \varnothing$$

Las expresiones restrictivas, pues, tienen forma clausal y de modo que cada cláusula contiene un solo literal positivo. Este literal positivo forma la *cabeza* de la cláusula, y el resto de literales de la disyunción constituye el *cuerpo* de la cláusula. Para hacer referencia separadamente a ambos componentes se utilizarán las funciones siguientes:

*head*(p), que al operar sobre una cláusula p devuelve la cabeza de dicha cláusula.
*body*(p), que al operar sobre una cláusula p devuelve el cuerpo de dicha cláusula.

Asimismo, se define *Horn*(p, q), que al operar sobre una cláusula de tipo hecho y una cláusula de tipo objetivo devuelve la cláusula de tipo regla que resulta de combinar ambas. Evidentemente,

$$head(Horn(p, q)) = p$$
$$body(Horn(p, q)) = q$$

**Requisitos**

Los requisitos hacen referencia sólo a los sumideros, es decir, son predicados en los cuales los únicos átomos que aparecen son sumideros del componente en cuestión. Un requisito toma la forma de una cláusula Horn que sólo tiene cuerpo, es decir, es una cláusula de tipo *objetivo*:

$$\text{Dado el requisito p, } head(p) = \varnothing$$

Además, cada requisito está asociado a un sumidero, y cada sumidero está asociado a un requisito como máximo. Se define la función *rAsoc*, que dado un sumidero de cierto componente de cierto sistema devuelve un conjunto con un único elemento que es el requisito asociado a tal sumidero. (Si un sumidero no tiene explícitamente asociado su requisito, por defecto se asume que tal requisito es el predicado `true`).

$$rAsoc : \mathbf{C} \times \mathbf{A} \rightarrow \mathbf{P}^1$$

$$k_i \in K(C) \Rightarrow rAsoc(C, k_i) = \{p_i(k_1, k_2, ..., k_m)\},\ p_i(k_1, k_2, ..., k_m) \in R(C)\ ,\ \text{card}(rAsoc(C, k_i) = 1$$

$$\forall\ p_i(k_1, k_2, ..., k_m) \in R(C)\ \exists!\ k_i \in K(C)\ /\ rAsoc(C, k_i) = \{p_i(k_1, k_2, ..., k_m)\}$$

Nótese que esto no impide que existan diferentes $p_i(k_1, k_2, ..., k_m)$ y $p_j(k_1, k_2, ..., k_m)$ que sean equivalentes; lo que importa es que la función *rAsoc* permite identificar el predicado asociado a un sumidero, independientemente de cuál sea su "parecido" con otros predicados (parecido que puede llegar hasta la equivalencia lógica).

De este modo, puede ofrecerse una definición "constructiva" de R(C):

$$R(C) = \cup_{i=1...m}\ rAsoc(C, k_i)$$

Es decir, R(C) es en realidad el conjunto formado por la unión de todos los requisitos asociados a los sumideros del componente C.

Cuando pueda deducirse fácilmente por el contexto, se suprimirá en la llamada a la función el parámetro C.

**Garantías**

Las garantías son expresiones restrictivas que hacen referencia tanto a los sumideros como a las fuentes. Cada garantía está asociada a una sola fuente, aunque a diferencia de lo que ocurría entre restricciones y sumideros, una fuente puede estar asociada a muchas garantías. Se define, pues, la función *gAsoc*:

$$gAsoc : \mathbf{C} \times \mathbf{A} \rightarrow \mathbf{P}^n$$

$$s_i \in S(C) \Rightarrow gAsoc(C, s_i) = \{q_j(k_1,..., k_m, s_1,..., s_n)\ /\ q_j(k_1,..., k_m, s_1,..., s_n) \in G(C)\}$$

Esto permite realizar una definición de G(C) análoga a la vista para R(C):

$$G(C) = \cup_{i=1...n} gAsoc(C, s_i)$$

**Dependencia de cláusulas**

Dado un conjunto de cláusulas H, diremos que una cláusula p depende de otra q (y se denotará por p ::- q) si:

$$\exists\ p_b \in body(p)\ /\ p_b = head(q)$$

Es decir, p ::- q si para averiguar el valor de verdad de *head*(p) es necesario averiguar el valor de verdad de *head*(q).

La negación de la relación de dependencia se denota por ::~.

### 4.2.3. Sistema

Un sistema $\Omega$ es un grafo finito (cuyos nodos $\nu$ son componentes y cuyos arcos $\varepsilon$ son pares fuente / sumidero), al que se añade un conjunto $\boldsymbol{L}$ de predicados auxiliares (que también son cláusulas Horn). En este caso se utilizará la notación de punto para hacer referencia a los puntos de conexión (ya sean fuentes o sumideros) de cada componente, como sigue:

$$\Omega = \{\nu, \varepsilon, \boldsymbol{L}\}$$
$$\nu = \{C_1, \ldots, C_n\}$$
$$\varepsilon = \{(C_i.s_j, C_k.k_l)\}$$
$$\boldsymbol{L} \subset \mathbf{P}$$
$$\forall\ p \in \boldsymbol{L},\ p \text{ no hace referencia a ningún punto de conexión}$$

También se utilizarán las notaciones $\nu(\Omega)$, $\varepsilon(\Omega)$ y $\boldsymbol{L}(\Omega)$ para hacer referencia a los nodos (componentes), los arcos (conexiones entre fuente y sumidero) y los predicados auxiliares de un sistema dado $\Omega$, respectivamente.

Se definen las funciones de acceso a los componentes de un arco. Dado $e = (C_i.s_j, C_k.k_l) \in \varepsilon(\Omega)$,

$$from(e) = C_i.s_j$$
$$to(e) = C_k.s_l$$

El dominio de todos los sistemas posibles se denota por **S**. Asimismo, mientras que **P** y **A** son respectivamente los dominios de los predicados y de los átomos (y por tanto conjuntos infinitos), P y A son los conjuntos de todos los predicados y átomos utilizados en un sistema dado (y por tanto conjuntos finitos). También se denotan por $P(\Omega)$ y $A(\Omega)$.

**Corrección topológica**: Un sistema $\Omega$ se dice topológicamente correcto (y se denota con el predicado $Tc(\Omega)$) si no hay puntos de conexión aislados y no hay ningún arco repetido:

$$Tc(\Omega) \Leftrightarrow$$
$$\forall\ C_i \in \nu(\Omega), \forall\ s_j \in S(C_i)\ \exists\ (C_i.s_j, C_k.k_l) \in \varepsilon(\Omega) \wedge$$
$$\forall\ C_k \in \nu(\Omega), \forall\ k_l \in S(C_k)\ \exists\ (C_i.s_j, C_k.k_l) \in \varepsilon(\Omega) \wedge$$
$$\forall\ e_i \in \varepsilon(\Omega)\ \mathrm{A}\ e_j \in \varepsilon(\Omega)\ /\ \ e_i = e_j$$

## 4.2.4. El modelo de verificación

### Base de conocimientos en bruto

La *base de conocimientos en bruto* para $\Omega$ (denotada como $\boldsymbol{K}_r(\Omega)$) es el conjunto de todas las expresiones restrictivas de todos los componentes del sistema (suponiendo que tanto fuentes como sumideros se mencionan con su nombre calificado para evitar ambigüedades) más todas las expresiones auxiliares. Los subconjuntos de requisitos y garantías se mantienen separados y por tanto identificables:

$$\boldsymbol{K}_r : \mathbf{S} \rightarrow \mathbf{P}^n \quad \boldsymbol{R}_r : \mathbf{S} \rightarrow \mathbf{P}^n \quad \boldsymbol{G}_r : \mathbf{S} \rightarrow \mathbf{P}^n$$

$$\boldsymbol{R}_r = \lambda\Omega \,.\, \{p \mid p \in R(C_i),\, C_i \in \nu(\Omega)\}$$
$$\boldsymbol{G}_r = \lambda\Omega \,.\, \{q \mid q \in G(C_i),\, C_i \in \nu(\Omega)\}$$
$$\boldsymbol{K}_r = \lambda\Omega \,.\, \boldsymbol{R}_r(\Omega) \cup \boldsymbol{G}_r(\Omega) \cup \boldsymbol{L}(\Omega)$$

La base de conocimientos en bruto contiene todo el conocimiento sobre el comportamiento de cada uno de los componentes del sistema. Pero presenta importantes limitaciones, porque no incorpora ninguna información sobre la interconexión entre los componentes. Nótese que aunque la definición de $\boldsymbol{K}_r$ indica que se parte de un sistema para llegar a un conjunto de predicados, lo cierto es que $\boldsymbol{K}_r$ depende sólo de $\nu(\Omega)$ y de $\boldsymbol{L}(\Omega)$, y en su definición no se hace referencia alguna a $\varepsilon(\Omega)$. Esto implica que la información sobre los arcos del grafo de componentes no interviene en la formación de $\boldsymbol{K}_r$ y por tanto cualquier otro sistema con los mismos componentes conectados de manera completamente distinta dará como resultado la misma $\boldsymbol{K}_r$.

El proceso constructivo que se define a continuación tiene como fin superar esta limitación y generar una base de conocimientos que contenga implícitamente la información sobre la topología del sistema.

### Renombramiento de átomos

**Renombramiento de átomos**: La función de sustitución de átomos *subs* se define como sigue:

$$subs : \mathbf{P} \times \mathbf{A}^n \times \mathbf{A}^n \rightarrow \mathbf{P}$$
$$subs(p(a_1, \ldots, a_n), \{a_i, ..., a_j\}, \{b_i, ..., b_j\}) = \sigma(p(a_1, \ldots, a_n))$$

donde $\sigma = \{a_i / b_i, ..., a_j / b_j\}$ es la sustitución que, aplicada a cualquier predicado, produce un nuevo predicado en el cual todas las ocurrencias de cada $a_i$ han sido reemplazadas simultáneamente por el correspondiente $b_i$.

Asimismo, se define una función *subsn* que puede aplicarse a un conjunto de predicados:

$$subsn : \mathbf{P}^n \times \mathbf{A}^n \times \mathbf{A}^n \rightarrow \mathbf{P}^n$$
$$subsn = \lambda P \lambda a \lambda b \,.\, \{q \mid p \in P,\, q = subs(p, a, b)\}$$

La función de generación de alias para átomos, *alias*, se define como sigue:

$$alias : \mathbf{P}^n \rightarrow \mathbf{A}$$
$$alias = \lambda P \,.\, b \in \mathbf{A} \,/\, b \text{ no aparece en ningún predicado de } P$$

Se define también una función de generación de nombres de átomo a partir de un arco del sistema. Esta función generará un átomo único para cada arco (y siempre el mismo).

$$genAtom : \mathbf{A} \times \mathbf{A} \rightarrow \mathbf{A}$$

$$genAtom = \lambda e \,.\, x \,/\, \forall\, e_i, e_j \in \varepsilon(\Omega)\ i \neq j \Rightarrow genAtom(e_i) \neq genAtom(e_j) \wedge genAtom(e_i) \notin A(\Omega) \wedge genAtom(e_j) \notin A(\Omega)$$

**Base de conocimientos**

**Definiciones preliminares:** Se define la función *and*:

$$and : \mathbf{P}^n \rightarrow \mathbf{P}$$
$$and = \lambda p_1 \lambda p_2 ... \lambda p_n \,.\, p_1 \wedge p_2 \wedge ... \wedge p_n$$

**Proceso de construcción:** Sea $\varepsilon(\Omega) = \{e_1, e_2, \ldots, e_n\}$ el conjunto de arcos de un sistema dado $\Omega$ y supóngase que se verifica $Tc(\Omega)$. Como se ha descrito previamente, cada elemento de $\varepsilon(\Omega)$ tendrá la forma $(C.s_i, D.k_j)$ donde C y D son componentes, $s_i$ es una fuente de C y $k_j$ es un sumidero de D.

Definimos entonces un proceso de construcción. Se comienza por definir cómo se obtiene la nueva restricción de un sumidero, $rAsoc'(k_i)$, mediante la combinación de varias expresiones a través de un operador de conjunción:

Dado $\{p(k_1, ..., k_m)\} = rAsoc(k_i)$ (conjunto que en este caso tiene cardinalidad 1), se define $Y(k_i) = \{p' \,/$

$$to(e_a) = k_1, ..., \; to(e_b) = k_m,$$
$$p' = subs(\; p(k_1, ..., k_m), \{k_1, ..., k_m\}, \{genAtom(e_a), ..., genAtom(e_b)\} \;)$$

$\}$

y partiendo de esta definición del conjunto Y, se define

$$rAsoc'(k_i) = and(Y(k_i))$$

Con lo que sólo resta definir el conjunto de restricciones de la base de conocimientos definitiva:

$$\boldsymbol{R}(\Omega) = \cup\, rAsoc'(Ci.k_j)$$

De cara a las garantías, se sigue un procedimiento similar. La diferencia es que las garantías son cláusulas Horn que pueden tener cabeza y cuerpo, y esto exige un tratamiento específico que se justificará más adelante.

Dado $\{q(k_1, ..., k_m, s_1, ..., s_n)\} = gAsoc(s_j)$, se define

$Z_{body}(s_i, q_j) =$ { qb' /

$to(e_a) = k_1, ..., to(e_b) = k_m, from(e_c) = s_1, ..., from(e_d) = s_n,$

$qb' = body(subs\ (q_j(k_1, ..., k_m, s_1, ..., s_m),$

$\{k_1, ..., k_m, s_1, ..., s_m\},$

$\{genAtom(e_a), ..., genAtom(e_b), genAtom(e_c), ..., genAtom(e_d)\}$

$)$

}

$Z(s_i) =$ { $Horn(qh, qb)$ /

$to(e_a) = k_1, ..., to(e_b) = k_m, from(e_c) = s_1, ..., from(e_d) = s_n,$

$q \in gAsoc(s_i),$

$qh = head(subs\ (q),$

$\{k_1, ..., k_m, s_1, ..., s_m\},$

$\{genAtom(e_a), ..., genAtom(e_b), genAtom(e_c), ..., genAtom(e_d)\}$

$),$

$qb = and(Z_{body}(s_i, q))$

}

Con esto, las garantías asociadas a una fuente dada son

$$gAsoc'(s_i) = Z(s_i)$$

y el conjunto de garantías de la base de conocimientos definitiva:

$$\boldsymbol{G}(\Omega) = \cup gAsoc'(Ci.s_j)$$

La base de conocimientos definitiva, finalmente, es:

$$\boldsymbol{K}(\Omega) = \boldsymbol{R}(\Omega) \cup \boldsymbol{G}(\Omega) \cup \boldsymbol{L}$$

Nótese que $\boldsymbol{L}$ permanece igual que al principio. Puesto que todas las sustituciones aplicadas afectan a puntos de conexión, y los predicados de $\boldsymbol{L}$, por definición, no hacen referencia a ningún punto de conexión, no es necesario modificar $\boldsymbol{L}$.

$\boldsymbol{K}$ contiene el conjunto de todos los requisitos y garantías de los componentes individuales, transformados de manera que la información sobre conexiones está implícita (gracias a la sustitución de fuente y sumidero por un nuevo alias que representa a la conexión entre ambos) y teniendo en cuenta que las expresiones restrictivas se *copian* la veces necesarias para reflejar las diferentes combinaciones en el caso de conexiones múltiples.

Por lo que se refiere a las conexiones múltiples, el proceso descrito conduce a que un requisito de un punto de conexión se convierte en la combinación conjuntiva del mismo requisito modificado para hacer referencia a cada una de las conexiones vigentes sobre ese punto. Por su parte, cada garantía de una fuente se repite tantas veces como conexiones hay vigentes en esa fuente, pero el cuerpo de la regla se convierte en una conjunción de todas

las posibles combinaciones de las referencias de ese cuerpo a las diferentes conexiones establecidas sobre los sumideros del componente.

## 4.3. Cuestiones de implementación

A continuación se van a comentar algunos aspectos de la relación existente entre el modelo en un sentido abstracto y su implementación.

### 4.3.1. Lógica de primer orden

Las expresiones restrictivas que reúnen el conocimiento que se tiene sobre cada componente toman la forma de cláusulas Horn. Esto hace que muchas cuestiones de implementación puedan responderse basándose en la experiencia que ya existe en la implementación de otros sistemas informáticos basados en el mismo concepto [EK76].

De cara a la **representación** en la práctica de estas cláusulas Horn, parece una solución evidente recurrir a un lenguajes de programación basado en dicha estructura, que ya ofrece una notación bien definida y no ambigua, una notación que además puede editarse por medio de editores de texto convencionales. La programación lógica declarativa, y en concreto el lenguaje Prolog [CM81], parece un vehículo apropiado.

Además de la mera representación de ese conocimiento, está claro que hay que realizar **procesos de inferencia** sobre esas reglas. El propio lenguaje Prolog incluye un motor de inferencia, por lo que nuevamente parece adecuado basarse en lo que este sistema ofrece en lugar de reinventar una notación y un modelo de inferencia, que además acabaría pareciéndose notablemente a este.

### 4.3.2. Alusiones a los puntos de conexión

Al redactar las expresiones restrictivas, puede comprobarse en el modelo que es necesario incluir átomos que representan a las fuentes o sumideros del componente. Para realizar el proceso de sustitución descrito en 4.2.4, sería necesario posteriormente tener la capacidad de identificar estos átomos.

Esto puede hacerse de muchas formas; por ejemplo, una posibilidad es recurrir a la descripción del componente en el momento oportuno, y comprobar si los nombres de átomo correspondientes a las fuentes o sumideros aparecen en la regla tratada. Otra opción puede ser simplemente considerar el texto de las reglas como "plantillas" en las cuales las alusiones a las fuentes y sumideros se encuentran delimitadas por caracteres especiales, como "$"; esta vía de actuación es menos rigurosa y aumenta la probabilidad de que se produzcan errores, aunque es más fácil de implementar (y así se ha obrado en los prototipos aquí desarrollados). Existen más alternativas; en cualquier caso, el problema es relativamente sencillo y se trata de una decisión de implementación que no tiene implicaciones importantes respecto al modelo, siempre y cuando permita llevar a cabo los algoritmos de generación.

### 4.3.3. Programación Lógica con Restricciones (CLP)

Al realizar los primeros experimentos de implementación en esta tesis, se encontró una notable deficiencia del lenguaje Prolog. Este lenguaje basa su proceso de inferencia en la

realización de unificaciones, que en sucesión conducen a la verificación o rechazo del objetivo o *goal*. Pero dicho proceso de unificación no tiene en cuenta el tratamiento de dominios. Es decir, Prolog presenta ciertas limitaciones en su proceso de unificación que pueden resultar relevantes para el propósito de Itacio.

Por ejemplo, si se exige como requisito que un valor sea mayor que 0, y se tiene como hecho cierto (garantía) que dicho valor es mayor que 5, resulta evidente que el requisito se cumple. Pero Prolog no dispone de mecanismos implícitos, en su proceso de unificación, para unificar ambas afirmaciones y dar por cierto el objetivo.

Esto puede considerarse una limitación importante. Habría que recurrir (y en algunos experimentos así se ha hecho) a construcciones que permitan obtener resultados parecidos; definiendo predicados tales como `es_positivo/1`, `mayor_que/2`, `mayor_o_igual_que/2`, etc. etc. y diversas relaciones entre ellos. Especialmente a nivel de microcomponentes, es muy frecuente que los requisitos se establezcan a nivel de valores y hagan referencia a cuestiones de dominios. Por poner un ejemplo, el fracaso del cohete Ariane V se debió precisamente a un problema de dominios incompatibles no detectado que se manifestó durante el vuelo [Li96]. Los rangos de valores posibles de las variables (cuestiones, al fin y al cabo, de dominio) constituyen una fuente tradicional de problemas.

Una tecnología que puede llenar este vacío del Prolog tradicional es la Programación Lógica con Restricciones (CLP en inglés) [Fr93]. En realidad, la CLP no pretende ser sólo un sistema de "Prolog extendido" para manejar dominios, sino algo más genérico y potente, un sistema de resolución de problemas acotados por ciertas restricciones.

CLP es un tipo de lenguajes de programación que combinan los aspectos declarativos de la programación lógica con la eficiencia de la resolución de restricciones. Se aplica a diversos tipos de problemas, desde los más típicos de búsqueda combinatoria como la planificación, gestión de horarios o asignación de recursos, que en principio no era práctico abordar con la programación lógica tradicional, hasta el análisis de circuitos [SGJ95]. Los lenguajes de CLP son especialmente eficientes, y su propósito es que, ofreciendo una eficiencia comparable a la de los lenguajes imperativos en este tipo de problemas, permiten un ciclo de desarrollo mucho más corto.

El enfoque utilizado para resolver problemas con CLP guarda cierto parecido con el de la investigación operativa tradicional. Los pasos seguidos para resolver un problema son los siguientes:

- Analizar el problema a resolver, para comprender con claridad cuáles son sus partes.
- Determinar las relaciones y condiciones aplicables entre las partes.
- Establecer, en forma de ecuaciones, dichas relaciones y condiciones. Para ello, es necesario elegir las variables y relaciones apropiadas. Además, la CLP suele ofrecer diferentes sistemas de restricciones, que serán más o menos apropiados dependiendo del caso.
- Resolver estos sistemas de ecuaciones. Los sistemas de CLP son, de hecho, sistemas de resolución automática, por lo cual esta fase es transparente para el usuario.

La aportación de la CLP frente a la investigación operativa es que en la CLP se dispone de diferentes dominios para las restricciones, y en que estas restricciones se pueden generar de

forma dinámica, por lo que la resolución de ecuaciones se ve enriquecida por una faceta de programación. En la CLP el lenguaje en cuestión incorpora algoritmos eficientes para resolución, búsqueda, etc.

De este modo, la CLP permite expresar condiciones sobre las variables que se desee, con lo que la tradicional laguna del Prolog a la hora de unificar condiciones referidas a dominios o rangos se subsana. Además, el tipo de restricciones no se limita a la verificación de rangos, sino que se extiende para poder resolver otro tipo de problemas, como por ejemplo problemas de ordenación. Por otra parte, merece la pena hacer notar que en general los sistemas de CLP resultan mucho más flexibles y dinámicos que sus equivalentes algorítmicos tradicionales, pero al mismo tiempo suelen ser bastante eficientes. Los algoritmos que utilizan internamente están muy optimizados, y suelen ofrecer muy buenos resultados en cuanto a tiempo de ejecución.

En ocasiones, un sistema de CLP, tras los sucesivos pasos de refinamiento y unificación, no es capaz de llegar a una solución concreta; en estos casos, el resultado final es, en sí mismo, un conjunto de restricciones, llevadas hasta el grado de concreción que se puede alcanzar con la información disponible. Este es un resultado perfectamente legal para muchos propósitos, y puede ser así en el modelo Itacio. Un sistema de CLP, programado bajo esta interpretación, podría ofrecer restricciones, es decir, especificaciones que debe cumplir determinado componente. Además, lo habitual es que, una vez obtenidas las restricciones a las que se puede llegar, un sistema CLP esté capacitado para ofrecer al usuario una búsqueda cuyo resultado será el conjunto de todos los elementos del dominio que cumplirían las restricciones señaladas. La solución a un problema de CLP es, con frecuencia, un subdominio, un conjunto designado por comprensión (que mediante sencillas primitivas del lenguaje puede también obtenerse en su versión por extensión).

Estas características de CLP la hacen especialmente indicada para servir de soporte a Itacio. La unificación entre dominios ya es de gran utilidad, dada la frecuencia con la que se establecen restricciones sobre rangos de valores y similares; pero el enorme potencial de CLP para plantear restricciones y resoluciones mucho más complejas (y que en la aplicación experimental de Itacio no se ha explorado) parece un recurso muy valioso.

### 4.3.4. La Hipótesis del Mundo Cerrado (CWA)

Dado que las expresiones restrictivas se asemejan a enunciados de lógica proposicional, y dado que se considera el conjunto de dichas expresiones restrictivas como una base de hechos y reglas sobre la que se aplicarán procesos de inferencia lógica, se plantean interrogantes respecto al modo en que se adoptarán decisiones basadas en esta información, tanto la establecida en forma de hechos como la obtenida mediante procedimientos deductivos. Y un capítulo muy importante del modelo Itacio es la llamada Hipótesis del Mundo Cerrado, que en adelante se denotará como CWA (*Closed World Assumption*).

Básicamente, la Hipótesis del Mundo Cerrado es una regla fundamental que rige el proceso de inferencia en algunos sistemas deductivos, como el incorporado en el lenguaje de programación Prolog. Dado un programa en Prolog (o lo que es lo mismo, dada una base de hechos y reglas) y dada la pregunta-objetivo que se plantea a dicha base (el llamado *goal*) el proceso de inferencia puede, en ocasiones, llegar a deducir que la expresión-objetivo es simplemente cierta o, si el objetivo incluye variables libres, deducir posibles valores de dichas variables que hacen que la expresión-objetivo sea cierta. Esta es la potencia de los

sistemas de programación declarativa o basados en conocimiento. En ocasiones, el sistema puede deducir también lo contrario: puede encontrar evidencias de que el enunciado es explícitamente falso.

Pero ¿qué ocurre si el sistema de inferencia no encuentra evidencias de que la afirmación planteada sea cierta? El razonamiento humano establece una distinción entre el caso de encontrar evidencias de que algo es falso y el caso de no tener información que asegure si es falso o es cierto. Pero los sistemas de razonamiento automático como Prolog suelen igualar ambos casos y acogerse a la CWA: todo aquello que no se tenga constancia de que es cierto, será considerado falso. Dicho de otro modo, el mecanismo de inferencia se ciñe exclusivamente a la información de que dispone, y asume que no hay nada más allá de eso.

De alguna forma, la CWA es una suposición pesimista que permite que los sistemas de inferencia que la utilizan se enfrenten a un problema de tamaño limitado y perfectamente conocido (que no es más que la base de hechos y reglas de que se dispone). Pero la CWA, en su traslación al sistema Itacio, que no es un sistema de inferencia sino de verificación de componentes, juega un papel importante que requiere más explicaciones.

En Itacio, el propósito de la CWA no es sólo acotar el problema de inferencia que plantea la verificación de los componentes. En este modelo se ha tomado la decisión de que las expresiones restrictivas (ya sean requisitos o garantías) son expresiones que se pueden evaluar como ciertas o falsas, sin posibilidad de equívoco y sin grados intermedios, de modo que la interoperabilidad de dos componentes se pueda evaluar también de forma inequívoca verificando la satisfacción de los requisitos a partir de las garantías. La CWA equivale a estrechar aún más el cerco sobre los defectos, de forma que si no se tiene constancia de que los requisitos se cumplen, se asume que no se cumplen.

Somos conscientes de que esto plantea un alto nivel de exigencia de orden práctico, ya que muchas de las combinaciones de componentes que de manera habitual serían válidas se convierten en ilegales simplemente porque no se tiene constancia explícita de que las condiciones de funcionamiento sean adecuadas en todos los casos posibles. En Itacio, nunca es válida una invocación de la función "raíz cuadrada" pasando un parámetro que "en general" es positivo. Esa invocación sólo será legal si se tiene la absoluta seguridad de que, de acuerdo con las especificaciones de los componentes involucrados, dicho parámetro es siempre positivo (salvo, claro está, un incumplimiento involuntario de alguna especificación debido a defectos de orden interno en un componente).

Esto representa un cierto choque respecto a la forma tradicional de programar; no es frecuente que en la práctica se comprueben *todos* los valores de retorno de error o *todos* los posibles rangos de variables. No es raro que los programadores verifiquen las condiciones de error más probables, pero supongan unas condiciones de funcionamiento "normales", y si esas condiciones dejan de cumplirse, se produzca una situación excepcional que se tratará como buenamente se pueda (y el tratamiento de excepciones es un problema de gran envergadura).

La CWA aplicada en Itacio a través de CLP pretende luchar contra este enfoque, asumiendo que el software debe ser (hasta donde es posible asegurar) correcto por construcción. Itacio no garantiza la inexistencia de defectos, pero acogiéndose a la CWA al menos se descarta ensamblar dos componentes que no vean satisfechos todos y cada uno de los requisitos que explícitamente plantean. Y por supuesto no se asume un "margen habitual de funcionamiento", a no ser el que los propios componentes garanticen y

documenten explícitamente. El no saber si un componente producirá fallos es "moralmente equivalente" a tener constancia de que lo hará.

A la hora de utilizar un componente que deja alguna restricción sin satisfacer, habrá que recurrir a componentes adicionales, que hagan el papel de adaptador y aporten las garantías necesarias. Estos componentes serán quizás funcionalmente equivalentes a una verificación en tiempo de ejecución y a un tratamiento de condiciones de error, pero al menos:

- Se tendrá una percepción clara de dónde y cómo se está realizando esa tarea.
- Se podrá normalizar el tratamiento de dichas condiciones excepcionales.
- Se habrá tomado una decisión explícita de tipo funcional sobre qué ocurre en el programa cuando se dan esas condiciones. Estos defectos no estarán ocultos ni protegidos por la incertidumbre.

No se descarta (véase el capítulo 6) que el modelo Itacio se vea ampliado para manejar razonamiento aproximado (al estilo de sistemas de diagnóstico ya tradicionales en Inteligencia artificial, como PROSPECTOR [DGH79] y MYCIN [Sh79]) y ofrecer resultados cuantitativos sobre la idoneidad de una conexión o su índice de viabilidad. Pero esto no anularía el papel de la CWA; la incertidumbre estaría controlada y seguiría estando sujeta a lo marcado por los hechos y reglas de la base de conocimientos, que simplemente darían lugar a razonamiento aproximado.

### 4.3.5. Representación de un sistema

Llegados a este punto, parece claro que cada componente lleva asociadas de alguna forma sus expresiones restrictivas. Pero una mera colección de componentes no es suficiente para realizar la verificación de un sistema ni para modelarlo; es necesario describir las diferentes *instancias* de cada componente que forman parte del sistema que se está construyendo, y también la forma en que dichas instancias se conectan entre sí. Es decir, como se ha visto anteriormente el sistema viene dado por:

$$\Omega = \{\nu, \varepsilon, \boldsymbol{L}\}$$

donde $\nu$ es el conjunto de nodos (instancias de componentes), $\varepsilon$ es el conjunto de arcos (conexiones entre componentes) y $\boldsymbol{L}$ es la biblioteca de predicados de apoyo. Esta última consta de un conjunto de predicados similares a las expresiones restrictivas, pero que no hacen referencia a ningún componente concreto, sino que plasman conocimiento genérico; para su representación se puede recurrir al mismo lenguaje de tipo CLP que se haya utilizado para las propias expresiones restrictivas.

El conjunto de nodos y arcos, por su parte, no deja de ser un grafo; y un grafo puede representarse de maneras muy variadas. Al fin y al cabo, la representación de grafos no es ninguna novedad dentro de la Informática. En los experimentos realizados con Itacio, el grafo en cuestión se ha representado primero mediante un formato propietario de gráficos (véase capítulo 5.1.1), después mediante simples ficheros de texto (capítulo 5.1.2) y finalmente mediante una base de datos convencional (capítulo 5.1.3). La representación del grafo de componentes, pues, admite multitud de variaciones y no representa un problema especialmente importante desde el punto de vista técnico.

### 4.3.6. Generación de la Base de Conocimientos

Los algoritmos de generación de la Base de Conocimientos descritos en 4.2.4 se apoyan en el *renombramiento* de átomos, lo que equivale a una sustitución, y en la concatenación de los predicados resultantes. Dado que los predicados tienen forma textual (véase 4.3.1 y 4.3.3) el proceso de generación de la Base de Conocimientos requeriría simplemente implementar los algoritmos descritos y tratar las reglas mediante mera manipulación de cadenas de texto, lo cual no presenta grandes dificultades.

### 4.3.7. Predicados especiales

La Base de Conocimientos permite inferir el valor de verdad o falsedad de cada uno de los predicados que contiene. En realidad, lo que se verifica es la corrección de cada conexión, que viene dada por el cumplimiento de los requisitos asociados al sumidero de dicha conexión. Para realizar esta verificación, habrá que plantear al sistema el objetivo (*goal*) correspondiente.

Puede resultar conveniente reservar un predicado "estándar" para este fin, de modo que para comprobar la validez de una conexión se pueda invocar a un único predicado sobre el sumidero, que a su vez requiera el cumplimiento de las restricciones correspondientes. En el caso de los prototipos de Itacio así se ha hecho; basta con que el proceso de generación de la base de conocimientos tenga presente esta necesidad y genere estos predicados propios de Itacio. Por ejemplo, en el último prototipo de Itacio (véase capítulo 5.1.3) para cada punto de conexión a verificar se genera un predicado de la forma:

```
verify_connection(atomo_asociado_a_conexion) :- <requisitos...>
```

y un predicado general para todo el sistema, que es el que lanza el proceso de verificación:

```
verify_system :-
          verify_connection(punto_1),
          verify_connection(punto_2),
          ...
          verify_connection(punto_n).
```

Comentar que el algoritmo genera implícitamente la información sobre topología

### 4.3.8. Conexiones recursivas

Cabe dedicar cierta atención al problema de las *conexiones recursivas*. ¿Qué ocurre si un componente se conecta a sí mismo? ¿O si las conexiones forman un ciclo?

Considérese el caso básico de que un sumidero $k_1$ de un componente C esté conectado a una fuente $s_1$ del mismo componente. El modelo plantea simplemente la necesidad de que los requisitos del sumidero en cuestión se satisfagan; considérese el caso de que:

$$\nexists p \in rAsoc(C, s_1) \;/\; q ::\text{-}\; p,\ q \in gAsoc(C, s_1)$$

Dicho de otro modo, para resolver el valor de verdad de los requisitos del sumidero no es necesario resolver ninguna de las garantías de la fuente, porque los requisitos del sumidero no tienen ninguna relación de dependencia con las garantías de la fuente. En este caso,

aunque existe una recursividad "estructural" en las conexiones del componente dicha recursividad no tiene reflejo alguno en la estructura de reglas; no existe a efectos prácticos.

Sin embargo, sí puede darse el caso de que existan cláusulas p ∈ *rAsoc*(C, $k_1$) / q ::- p, q ∈ *gAsoc*(C, $s_1$), y a su vez p ::- q. Es decir, para resolver el valor de verdad del requisito, es necesario resolver el valor de verdad de la garantía de la fuente, y a su vez esa garantía requiere conocer el valor de verdad del requisito. En esta situación, la recursividad estructural tiene su reflejo en las dependencias entre las reglas, de forma que aparece una cadena de razonamiento circular.

Esto puede ser un problema o no serlo, como ocurre habitualmente con las estructuras recursivas. La cadena de inferencia recursiva puede encontrarse con un *caso base* que detenga la recursividad y permita finalizar el razonamiento, o puede ser realmente infinita, y en este caso el proceso de inferencia no se detendría. Pero sea cual sea el caso, este problema es igualmente posible en cualquier base de conocimientos programada fuera de Itacio; si se construyen predicados interdependientes de forma que la cadena de inferencia forme un bucle, y en dicha cadena recursiva de inferencia no se llega a un caso base, el proceso de inferencia se bloquea en un bucle infinito. Simplemente, se tendrá una base de conocimientos mal diseñada (o en este caso un componente incorrectamente conectado).

Los prototipos desarrollados como parte de esta disertación no detectan estos casos de "interbloqueo". Pero sería posible hacerlo; bastaría realizar un análisis de las dependencias entre reglas y comprobar que forman un grafo acíclico.

La conclusión es que las conexiones cíclicas entre componentes no representan un problema intrínseco de Itacio; es un problema que, en potencia, existe con cualquier base de conocimientos. Una conexión cíclica puede dar como resultado un proceso de inferencia infinito o no; se podría dotar al sistema de la funcionalidad para detectar estos problemas como paso previo a la verificación.

Finalmente, nótese que el propio motor de inferencia aquí utilizado, ECLiPSe, no realiza ese tipo de análisis y permite al usuario (sin estar Itacio involucrado) redactar bases de conocimientos con cadenas de inferencia cíclicas e infinitas; no cabe, pues, achacar a Itacio este problema como limitación intrínseca del modelo.

### 4.3.9. Asociación entre conocimiento y plataformas de componentes

En este punto se pueden hacer algunas consideraciones de orden práctico sobre la forma de asociar conocimiento a los componentes.

El conocimiento sobre los requisitos y garantías de un componente se recoge en expresiones restrictivas cuya estructura y características ya se han descrito en este capítulo y los anteriores. Si el modelo Itacio se está aplicando a un diseño, a entes teóricos, no hay mucho más que decir al respecto; se puede trabajar directamente con las expresiones que describen al componente. En el caso de los prototipos aquí desarrollados, así se ha hecho, y las expresiones restrictivas se han manejado por separado. La única conexión con el "mundo real" al que en cada caso representaban se establecía por parte del usuario, de manera implícita.

Si esas entidades, no correspondiendo a ninguna implementación, encuentran una representación real (por ejemplo, como elementos de cierta herramienta CASE) la misma

vía elegida para almacenar la información básica serviría para almacenar la información asociada a Itacio.

Existe otro caso a tener en cuenta; que las entidades manejadas sean componentes en sentido clásico, es decir, elementos de implementación (véase página 8). Esto implicaría probablemente manejar componentes de alguna de las plataformas de componentes descritas en el capítulo 2.9. En este caso, si el usuario está realmente ensamblando estos componentes, el que las especificaciones fuesen ligadas a los mismos de alguna forma tendría ventajas e inconvenientes.

Una ventaja clara de que las especificaciones fuesen parte intrínseca del módulo binario es que se ahorrarían problemas de distribución, y el componente sería algo más compacto. Además, sería posible que el entorno de programación en el que se están ensamblando los componentes manejase de manera directa estas especificaciones, con lo que el usuario, quizás en un entorno visual de composición, se limitaría a describir la estructura del sistema conectando los componentes, y vería el resultado de las validaciones.

Esto es técnicamente viable en prácticamente todos los casos. Las expresiones restrictivas no dejan de ser información textual, que cualquier componente puede contener. Un componente de una plataforma Windows, por ejemplo, puede contener esa información textual como parte de los llamados *recursos* [Pe98] de la DLL o módulo ejecutable. Sería posible establecer una interfaz genérica para que el entorno de composición pudiese acceder a esa información. Los JavaBeans también pueden contener información textual con facilidad, ya que los ficheros JAR pueden a su vez contener otros ficheros en su interior. En el peor de los casos, habría que almacenar esa información textual como datos estáticos del módulo, cosa también posible.

Un inconveniente de este enfoque es la otra cara de la ventaja mencionada: al formar el conocimiento y el componente una unidad compacta e indisociable, si se descubren errores en ese conocimiento (o simplemente se desea ampliarlo) no sería posible actualizar el conocimiento de manera separada y respetar el código ejecutable del componente. Sería necesario distribuir una nueva versión del componente completo.

Parece más aconsejable, pese a todo, distribuir los componentes con el conocimiento integrado, para evitar posibles inconsistencias entre el módulo binario y su descripción y para evitar las dificultades de gestión de versiones que implica su mantenimiento separado. Pero en cualquier caso se trata de una decisión de tipo práctico que no tiene gran relevancia por lo que se refiere al modelo teórico de Itacio o su viabilidad técnica.

## 4.4. Relación con los objetivos planteados

### 4.4.1. Componentes

Una de las ideas clave del modelo es que funcionalmente se considera a los componentes "cajas negras" con entradas y salidas. Los componentes son la unidad de composición que aquí se maneja, y lo que se evalúa en el proceso de verificación son las conexiones *entre* los componentes; el veredicto que el sistema ofrece indica si las conexiones realizadas son aceptables o no, y por qué.

Un primer motivo para considerar el uso de componentes y la definición que aquí se maneja es precisamente la simplificación que ofrece el encapsulamiento. Este modelo no

entra en cuestiones internas a cada componente; lo único que interesa es el comportamiento observable que el propio componente describe de manera explícita.

Básicamente, se da por supuesto que cada componente, por sí mismo y considerado de manera aislada, es correcto. Si este axioma se cumpliese, la única forma de cometer errores sería conectar ese componente con otros de manera incorrecta; es decir, ubicar ese componente en un entorno inadecuado. Puesto que el modelo Itacio permite precisamente verificar estas conexiones, se consigue centrar el problema; el modelo pretende detectar sólo los defectos que surgen de la combinación de componentes. Esto va en beneficio de la simplicidad, permite una cierta economía de conceptos y algoritmos, y por tanto de la asequibilidad que forma parte intrínseca de los requisitos impuestos por nuestra tesis.

¿Significa esto que se da por supuesto que cada componente es *realmente* correcto? Por supuesto que no necesariamente es así. Cualquier fragmento de software es un firme candidato a contener defectos. Pensando en componentes software según la definición habitual del término, está claro que la implementación plantea considerables retos de calidad, que deben abordarse con técnicas específicas. En primer lugar, la verificación "externa" de Itacio se vería complementada por muchos otros métodos (las pruebas unitarias tradicionales, otros métodos de verificación y validación, etc.) cuya misión sería detectar estos defectos que en principio Itacio considera inexistentes como axioma. Y en segundo lugar, *si los componentes están a su vez formados por subcomponentes, sería posible utilizar también Itacio dentro de cada componente, simplemente aplicándolo a un nivel de abstracción diferente*. Lo importante es que, en cada nivel de abstracción, Itacio centra su ámbito de actuación en las conexiones entre componentes, y esta simplificación juega un importante papel en la aplicabilidad de Itacio desde el punto de vista humano (facilidad de aprendizaje, economía de conceptos) y desde el punto de vista técnico (algoritmos más simples y robustos).

### 4.4.2. Expresiones restrictivas

La morfología de las expresiones restrictivas tiene relación con el análisis estático que se pretende realizar. Este análisis se basa en el *conocimiento* que a priori se tiene sobre la misión de un componente. Para describir un componente, es necesario escribir (en un lenguaje relativamente formal y susceptible de formar parte de un procedimiento automático, pero a la vez asequible) lo siguiente:

1. Las restricciones que el componente plantea a su entorno (entradas) como condiciones correctas de funcionamiento (qué *exige* el componente con respecto a sus entradas). Estas expresiones se llaman aquí *requisitos*.
2. Las expresiones que describen el comportamiento *observable* del componente a partir de las entradas (dicho de otro modo, *a qué se compromete* el componente con respecto a sus salidas). Estas expresiones se llaman aquí *garantías*.

Los dos tipos de expresiones utilizan una notación homogénea, de modo que pueden integrarse y ser tenidas en cuenta conjuntamente por el proceso de análisis. El conocimiento sobre requisitos y garantías se entremezcla y pone en relación mutua, de manera que es posible un proceso de inferencia que aproveche todo ese conocimiento.

Una cuestión esencial en la solución del problema es que se asume que este conocimiento sobre exigencias y garantías puede ser **erróneo**. Es decir, es posible –por error- que el comportamiento práctico de un componente no se ajuste a lo que las expresiones indican.

De aquí la expresión *hasta donde nos es posible asegurar* que figura en la versión simplificada de esta tesis (pág. 2).

Sería indudablemente un gran avance poder disponer de requisitos y garantías que se pudiese verificar que efectivamente responden a la realidad del código fuente al que se asocian. Existen, además, técnicas relacionadas con el problema de la relación entre los programas y sus especificaciones, y muchos métodos formales están destinados precisamente a ello [Mo94, AB92]. Pero este objetivo queda fuera del ámbito de esta tesis.

El compromiso que se adopta aquí consiste en convivir con la posibilidad de que el conocimiento que se tiene sobre un componente sea incorrecto, a cambio de poder expresar todo ese conocimiento (por imperfecto que sea) e integrarlo en un sistema de verificación automático, de modo que tal conocimiento nunca se desperdicie. De hecho, este conocimiento puede manejarse sin llegar a construir el componente, en etapas de diseño; y lo que es más, la idea genérica de componente que aquí se maneja puede ser aplicada a entidades que ni siquiera tienen relación directa con código fuente o programas ejecutables. El modelo puede utilizarse en un plano puramente teórico.

### 4.4.3. El modelo de verificación

El sistema elegido para recoger las exigencias y garantías es, como se ha dicho, la lógica de primer orden en forma de cláusulas de Programación Lógica con Restricciones (*Constraint Logic Programming*, CLP). Esto permite integrar todo ese conocimiento de manera homogénea y realizar deducciones mediante un motor de inferencias apropiado. Toda la tecnología involucrada en este proceso está sobradamente contrastada en su vertiente teórica y desarrollada en su vertiente práctica; además, es conocida en un grado razonable por los profesionales del desarrollo de software. Otros autores coinciden en que la programación lógica puede ocupar un lugar significativo en el desarrollo futuro de técnicas de ingeniería del software [Lau00, Mercury].

El porqué de la utilización de CLP en lugar de simple Prolog es que CLP permite realizar unificaciones e inferencias mucho más ambiciosas y complejas que el Prolog tradicional; se puede operar con dominios y no sólo con pura aritmética, y expresar restricciones mucho más potentes y versátiles. No obstante, para muchas aplicaciones no es necesario utilizar las extensiones de CLP y bastaría con las cláusulas tradicionales del Prolog.

La integración de las exigencias y garantías de un solo componente es inmediata, por simple yuxtaposición. Quedaría entonces el problema de la integración de las expresiones de los diferentes componentes de un sistema. Esto se ha logrado de una forma muy simple, mediante los algoritmos descritos en el capítulo 4, quedando la información sobre la topología del sistema recogida implícitamente en el conjunto de expresiones resultante.

Un sistema de inferencia convencional es capaz de deducir si determinada conexión entre dos componentes ve satisfechas las restricciones correspondientes, ya sea de forma directa (debido al comportamiento del componente vecino) o indirecta (como resultado del comportamiento combinado de diversos componentes). El proceso de inferencia utilizado por Prolog o CLP realiza las verificaciones *transitivas* que sea necesario, en una búsqueda en profundidad por las reglas y hechos recogidos de los diferentes componentes involucrados. Este proceso se remonta hasta donde sea necesario en la cadena de relaciones entre componentes. Además, la CWA contribuye a evitar ambigüedades y facilita la adopción de un enfoque conservador y defensivo hacia los defectos.

Con todo esto, se tiene un modelo que permite una verificación estática (basada en el conocimiento que a priori se tiene sobre un componente), no ambigua, automática (puesto que todo el proceso es plenamente automatizable y así se demuestra tanto en la concepción teórica basada en un proceso de inferencia convencional como en los prototipos), y asequible (las tecnologías involucradas están disponibles y han sido aplicadas con éxito en los prototipos, aun con recursos de desarrollo muy limitados).

# 5. Aplicación experimental del modelo

Como es lógico, las ideas presentadas hasta ahora debían ser validadas de alguna forma. Podían abordarse varios caminos para ello. En ocasiones, la demostración de una afirmación puede realizarse en un plano teórico, probablemente recurriendo a mecanismos matemáticos. En otros casos, se recurre a un procedimiento empírico que ofrezca cierto grado de confianza sobre la corrección de la teoría que se está sometiendo a prueba.

En el caso de esta tesis (orientada claramente a la ingeniería y por tanto ligada a la práctica), es dudoso que una demostración estrictamente teórica sea posible. Si se revisa la proposición que resume la motivación de este trabajo, que figura en el capítulo 1, se apreciará que al responder a tal proposición:

- No se está explicando un fenómeno ni realizando una afirmación matemáticamente demostrable: se está ofreciendo un método.
- Entre otros condicionantes, se exige para ese método que sea *asequible* (en el capítulo 1, pág. 3 se define con más claridad qué engloba ese término). En qué grado es asequible un método de trabajo es, de por sí, subjetivo, difícil de formalizar, difícil de cuantificar y evidentemente difícil de probar en un plano teórico.

Por tanto, parece claro que el camino para verificar que la solución ofrecida responde al planteamiento de la tesis es un camino empírico. Se discuten a continuación las características que la solución aportada debe cumplir y la forma posible de comprobar que las cumple. Como resultado de este análisis, se plantea la estrategia general de validación.

**La verificación del software debe ser estática**. Esto se puede demostrar con relativa facilidad. Por la propia construcción y definición del método, puede apreciarse si este:

a) requiere la construcción previa del sistema software que se desea verificar, o por el contrario

b) puede utilizarse a priori, independientemente de la existencia de dicho sistema software, basándose sólo en el conocimiento disponible.

Parece evidente que el sistema Itacio responde al supuesto b). En la definición del sistema de verificación (capítulo 4) no interviene el sistema software construido, ni se requiere la intervención de su código fuente u objeto; todo el proceso se basa en conocimiento declarativo. Por la propia definición de Itacio, pues, este objetivo parcial puede considerarse superado.

No obstante, la implementación de un sistema funcional, en la forma de un prototipo de viabilidad técnica, despejaría las posibles dudas al respecto.

**La verificación del software debe ser automática**. La situación es similar a la del subobjetivo anterior. En el capítulo 4 se ofrece un método constructivo susceptible de ser automatizado, y las expresiones restrictivas que son la base de Itacio se expresan en cláusulas Horn con sintaxis de Prolog. Estas expresiones, por diseño, son directamente manejables por parte de un sistema de Programación Lógica con Restricciones; dicho de otro modo, el conocimiento de Itacio se expresa en un lenguaje de programación, lo que despeja dudas sobre su tratamiento automático.

Al igual que en el caso anterior, el mostrar un prototipo de viabilidad de Itacio, con sus funciones íntegras aunque no tenga la calidad necesaria para su explotación, sería de gran ayuda para ganar confianza sobre el grado de consecución de este subobjetivo.

**Susceptible de ser implementado mediante técnicas conocidas y viables**. La forma lógica de abordar este punto parece ser describir de forma concreta qué técnicas pueden utilizarse para llevar a la práctica las ideas sobre Itacio, y evaluar si estas técnicas están lo suficientemente extendidas y consolidadas. Nuevamente, se trata de algo difícil de cuantificar, pero aunque haya un componente subjetivo, sí parece posible un cierto grado de acuerdo sobre la disponibilidad o no de una técnica.

**Susceptible de ser comprendido y aplicado en la práctica con relativa facilidad por personal de desarrollo**. Nuevamente, resulta muy difícil probar que un método es fácilmente aplicable por personal de desarrollo. Una demostración empírica podría requerir la aplicación real del método en diferentes empresas durante un tiempo significativo, y la comparación (de costes, tiempo de aprendizaje, calidad del software producido, etc.) con otras empresas que desarrollasen proyectos muy similares mediante otros métodos (por ejemplo, métodos formales tradicionales). Está claro que este tipo de ensayos no es viable en la práctica, por lo que no hay más remedio que realizar experimentos de ámbito más limitado. En esta línea, la realización de prototipos de viabilidad con medios de desarrollo muy limitados puede dar una idea de la dificultad relativa de uso del modelo.

**Flexible y susceptible de ser aplicado en diferentes ámbitos, no demasiado específico**. Con el fin de demostrar este aspecto, puede ser interesante la aplicación del sistema a diversos casos de estudio, muy diferentes entre sí. Si el sistema de verificación puede ser trasladado sin grandes modificaciones a estos problemas variados, puede considerarse razonable admitir que el método es, efectivamente, flexible.

## 5.1. Prototipos desarrollados

En este apartado se presenta una relación, y una somera descripción técnica, de las diversas implementaciones que se han ido realizando de prototipos de Itacio. Por supuesto, los prototipos en sí mismos no son el objetivo de la tesis; lo que se pretende es demostrar que es posible construir un sistema real utilizando técnicas estables y conocidas, es decir, que con recursos de desarrollo adecuados se puede crear un sistema de verificación real basado en Itacio.

Para cada prototipo se indicarán, pues, las bases técnicas y las características fundamentales.

### 5.1.1. Primer prototipo: Itacio-SEDA

#### Antecedentes

La idea de Itacio tiene su raíz en el trabajo profesional del autor en la empresa Seresco, S.A. desde abril de 1994. Tras algunos años en la empresa, la idea fue cuajando como resultado de los problemas cotidianos del desarrollo de software, y el primer documento escrito describiendo las ideas esenciales que vertebran el proyecto se crea el 20 de septiembre de 1997. En noviembre de 1999 comenzó la implementación del primer prototipo, que aquí se denomina Itacio-SEDA (véase Ilustración 14). Este fue el prototipo utilizado en los primeros experimentos que ayudaron a poner a prueba las ideas iniciales y concretar los algoritmos fundamentales.

Este prototipo se basa en el aprovechamiento de SEDA, una herramienta gráfica extensible propiedad de la empresa Seresco, S.A. [Seresco]. SEDA es en realidad una herramienta CASE que forma parte del sistema de desarrollo de software AIDA de esta misma empresa, proporcionando las funciones de edición gráfica e interacción con el sistema de información. Una de las principales características de SEDA es su extensibilidad; sobre un núcleo común de edición, es posible crear una estructura de *añadidos* que implementan el comportamiento de los diferentes tipos de diagramas y de objetos. Por ejemplo, SEDA ofrece soporte para la metodología OMT (diagramas de clases / objetos, diagramas de interacción, diagramas de transición de estados, etc.), metodologías de diseño de bases de datos (como Chen), metodologías de análisis estructurado (como Yourdon o Métrica) y otro tipo de recursos gráficos de uso general (como organigramas de flujo).

Además de ser utilizada en tiempo de desarrollo, SEDA puede también formar parte de aplicaciones terminadas, cuando estas requieran de edición gráfica. Por ejemplo, los sistemas de gestión de nóminas y personal de Seresco ofrecen la posibilidad de diseñar gráficamente (y depurar paso a paso) los algoritmos de cálculo de nómina, lo cual se ha conseguido también creando añadidos específicos que se integran con SEDA.

#### Estructura del prototipo

Seresco dio a mediados de 1999 su consentimiento para que SEDA fuese utilizada en la implementación del primer prototipo de Itacio. Este prototipo consistió, pues, en el desarrollo de un añadido de SEDA (llamado SEDAComp) que permitía manejar un nuevo tipo de diagrama, un diagrama de componentes. Los elementos de este diagrama eran componentes y conectores entre los mismos, reproduciendo directamente el modelo Itacio. Además de a las propiedades visuales de estos componentes, era posible acceder a las propiedades de validación, editando sus expresiones restrictivas asociadas. En resumen, el diagrama de SEDA contenía toda la información sobre un sistema que se quisiese modelar y verificar.

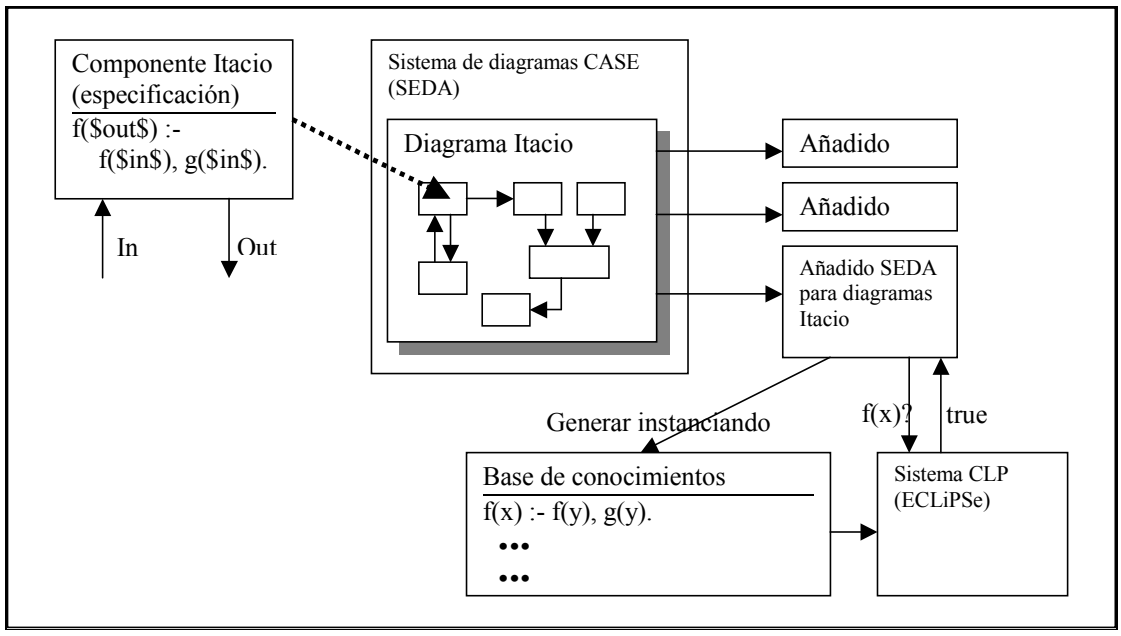



*Ilustración 14. Funcionamiento general del prototipo Itacio-SEDA.*

Para realizar la verificación, se decidió utilizar un sistema CLP ya existente, de modo que SEDAComp se comunicase con el motor de inferencia correspondiente, le suministrase la base de conocimientos generada y le interrogase, suministrándole los diferentes objetivos (*goals*). El sistema CLP elegido fue ECLiPSe [EC99, Wl97]. Este sistema comenzó a desarrollarse a principios de la década de 1990 en el ECRC de Munich, un centro propiedad de Bull, ICL y Siemens. Posteriormente, ECLiPSe fue transferido al Centre for Planning and Resource Control (IC-Parc) del Imperial College, que es la entidad que actualmente mantiene ECLiPSe.

ECLiPSe es un sistema de programación basado en Prolog, que pretende servir como plataforma de integración entre varias extensiones a la programación lógica, especialmente la CLP. El núcleo de ECLiPSe es una implementación especialmente eficiente del Prolog estándar (Prolog de Edimburgo) como se describe, por ejemplo, en [CM81]. La compilación se realiza en dos pasos, ya que primero se genera un código intermedio que se ejecuta a continuación en un emulador [Wa83]. ECLiPSe no es, en principio, un sistema comercial, y su distribución está limitada a instituciones educativas y de investigación.

La conexión entre SEDAComp y ECLiPSe requirió el desarrollo de un módulo de interacción, InterECL, puesto que la interfaz de programación de ECLiPSe tenía ciertos defectos que impedían su uso fiable.

Sobre este prototipo se hicieron pruebas relacionadas con la idea de los microcomponentes, y se llegaron a generar de forma automatizada programas elementales de cálculo aritmético. El usuario podía crear visualmente el programa en SEDA conectando los componentes involucrados, podía verificar la corrección del sistema, y posteriormente generar el equivalente en código C del mismo.

**Capacidades del prototipo**

- Edición gráfica de la estructura de un sistema.
- Edición de las expresiones restrictivas asociadas a cada componente.
- Un rudimentario sistema de generación de código, asociando a cada componente una "plantilla" que interviene en la generación (útil sólo para programas lineales muy sencillos).
- Verificación del sistema; el diagrama señala en rojo las conexiones que violan alguna restricción de funcionamiento, y el usuario puede pulsar sobre las mismas para conocer la razón del problema.

**Limitaciones más notables**

- Itacio-SEDA se basa en una primera versión del modelo Itacio. Esta primera versión no contemplaba la conexión de una fuente con más de un sumidero ni de un sumidero con más de una fuente.
- Esto implica que si una fuente debe conectarse a más de un sumidero, es necesario introducir un componente "duplicador", cuyas expresiones restrictivas "transmitan" el conocimiento de su entrada a sus salidas.
- Este primer diseño no permite diferenciar entre la definición de un componente y una instancia del mismo. Cada componente es un objeto independiente en un diagrama de SEDA; aunque esta herramienta sí permite crear características genéricas de los objetos de los diagramas, por limitaciones técnicas (debidas al uso "local" y limitado de SEDA) en este caso no es posible aprovechar esa capacidad. De este modo, si un componente aparece dos veces, hay que copiar y pegar las expresiones restrictivas asociadas al mismo.
- La instalación y configuración de este sistema plantea ciertas dificultades. Aunque el uso de SEDA permitió desarrollar una versión funcional en un espacio de tiempo muy breve, que era lo que se perseguía, planteaba ciertos problemas:
    - Existían evidentes limitaciones en la distribución del software (propiedad, en parte, de Seresco) y en la apertura del código fuente en caso de que llegase a ser necesario.
    - El uso aislado de SEDA no estaba previsto en la instalación original del producto. Por tanto, era necesario un ajuste manual para poder instalar el prototipo.
    - Aunque no se preveía que el código fuente resultase de especial interés en ninguna de las versiones de los prototipos, parecía sensato abordar una implementación más "ligera" e íntegramente desarrollada dentro del proyecto Itacio.

**Conclusiones**

Itacio-SEDA cumplió los objetivos marcados para su desarrollo. Permitió pulir las ideas iniciales sobre el modelo Itacio, concretando puntos que no estaban claros; permitió

también demostrar a efectos prácticos que en lo básico la implementación de Itacio como herramienta era posible con conocimientos y recursos de desarrollo muy limitados, y este desarrollo pudo hacerse además en un tiempo relativamente corto. Las conclusiones obtenidas de la construcción de este prototipo fueron refrendadas en ICSE 2000 [CLC00].

### 5.1.2. Segundo prototipo: Itacio-XJ

#### Antecedentes

Dada la situación del prototipo Itacio-SEDA, y teniendo en cuenta que había cumplido ya su misión y agotado su ciclo de aprovechamiento, se decidió avanzar en la implementación, y por las razones expuestas se decidió abandonar casi por completo Itacio-SEDA y comenzar desde cero el desarrollo de un prototipo que no dependiese de herramientas externas. El desarrollo del segundo prototipo comenzó a finales de marzo de 2000.

Esto supuso en principio la pérdida de la capacidad de edición gráfica. Se evaluó la posibilidad de construir el prototipo a base de JavaBeans y sobre el BeanBox [Kl99], que ya proporciona un sistema de edición en el que es posible ubicar y conectar componentes; pero se desechó esta posibilidad porque de todos modos el BeanBox no permite ver representadas gráficamente las conexiones entre los componentes (beans).

Por tanto, tras un breve estudio se decidió sacrificar la capacidad de edición gráfica; esta cuestión no era un objetivo de la tesis, y parecía demasiado costoso abordar el desarrollo o la integración de un editor gráfico, cuando Itacio-XJ iba a ser utilizado sólo con fines experimentales y la productividad derivada de su uso no tenía excesiva importancia. Se tomó la decisión de que toda la información estaría representada en ficheros de texto: tanto las descripciones de los componentes como la descripción de cada sistema (que haría referencia a los ficheros de componente mencionados). Cada "catálogo de componentes" podría ser un simple directorio del disco que contuviese las descripciones textuales de sus componentes.

Se decidió entonces abordar el desarrollo de Itacio-XJ utilizando Java para interpretar estos ficheros de texto.

#### Estructura del prototipo

La interfaz de usuario de Itacio-XJ es una interfaz web. En la idea de una implementación "ligera", sería posible incluso ejecutar el prototipo a distancia, puesto que el núcleo de la interfaz es un servidor web (en este caso Personal Web Server de Microsoft). La parte textual de la interfaz se basa en páginas HTML estáticas, haciendo uso de hojas de estilo (CSS).

En el caso de que se esté utilizando Itacio-XJ en una máquina distinta, sólo se pueden realizar consultas, pero si se está en la propia máquina servidora, la interfaz de usuario permite la edición de los ficheros de texto que contienen la información de los componentes y de los sistemas. Esta rudimentaria forma de edición de textos sustituye, pues, a la interfaz gráfica que proporcionaba Itacio-SEDA para crear los componentes y sistemas.

Una vez que existe algún sistema susceptible de ser analizado, el usuario puede solicitar la verificación del mismo. En ese caso, las páginas HTML ceden el control al generador de la

base de conocimientos, desarrollado en Java, y que se ejecuta en el servidor. Este conjunto de clases mimetiza el modelo Itacio (un sistema que consta de componentes, que a su vez tienen fuentes y sumideros, etc.), lo que le permite cargar toda la información de los ficheros de texto. Una vez que están en memoria los objetos necesarios, la información de las expresiones restrictivas se usa para generar la base de conocimientos de ese sistema en particular; esto también se hace reproduciendo el modelo, en este caso el algoritmo constructivo definido para Itacio en [CLC01a].

El resultado de esta generación se remite entonces a un sistema CLP, que es nuevamente ECLiPSe. De hecho, el componente InterECL del prototipo Itacio-SEDA se reutiliza en este nuevo prototipo, y vuelve a jugar el papel de interfaz con el motor de inferencia. Invocado desde Java, carga la base de conocimientos y va realizando las verificaciones que el núcleo Java le solicita.

Con estos resultados, la parte Java de Itacio-XJ puede hacer dos cosas: o bien generar para el navegador una página con la descripción textual del proceso de verificación (que en caso de errores explica qué falla y por qué), o bien generar una representación gráfica del sistema, que incluye marcas visibles para las conexiones no válidas. Es decir, aunque se renuncia a la capacidad de edición gráfica de los sistemas, sí se utiliza una *representación* gráfica de los mismos. Para ello, Itacio-XJ se apoya en una biblioteca ya existente de ubicación de grafos, GFC (*Graph Foundation Classes*) [Le97, Alpha].

Apoyándose en GFC para decidir la ubicación de los elementos gráficos, se genera un documento XML que contiene la información topológica del grafo y los detalles de la verificación. Este fichero XML se presenta gráficamente en el navegador web mediante una hoja de estilo XSL; el resultado de filtrar el XML a través del XSL arroja un fichero HTML cuyos gráficos están representados en el formato VML de Microsoft [VML]. Esto requiere el uso de un navegador que tenga instalado el soporte necesario para VML.

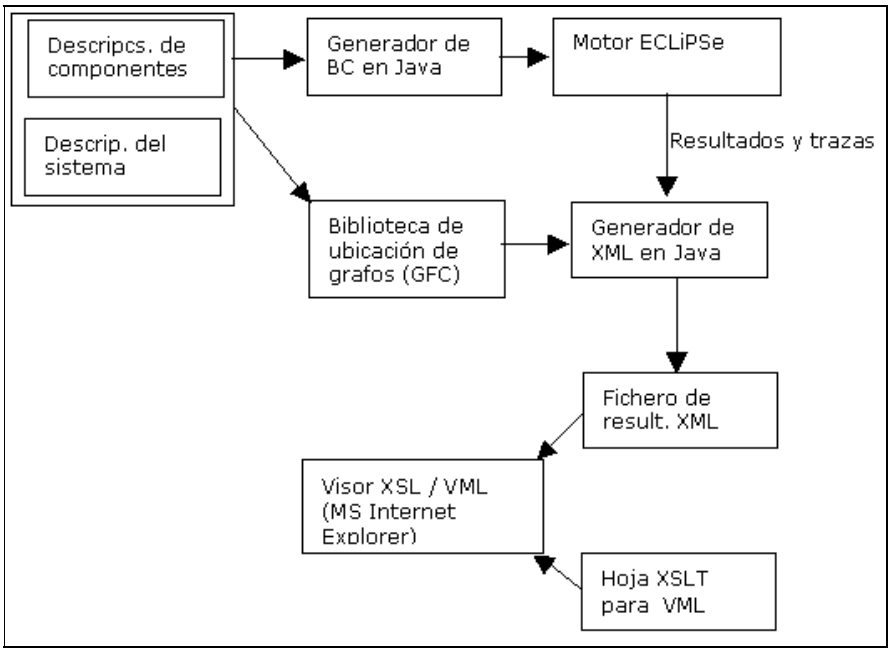



*Ilustración 15. Estructura del prototipo Itacio-XJ*

El resultado de esa validación es, pues, una representación gráfica del sistema; las conexiones que han violado alguna restricción aparecen señaladas en rojo y en gran tamaño, y cuando el usuario pulsa sobre las mismas puede ver una explicación sobre las restricciones incumplidas.

Como orientación sobre el tamaño de este prototipo, el total de código Java se reparte en 24 ficheros para un total de unas 3500 líneas de código, comentarios incluidos. El código xsl ronda las 160 líneas. El nombre "XJ" proviene del uso de estas técnicas (XML/XSL/Java).

**Capacidades del prototipo**

- El sistema se edita textualmente.
- La descripción de un componente puede reutilizarse en diversos sistemas o en muchas instancias dentro del mismo sistema.
- El prototipo se implementa mediante técnicas abiertas y ampliamente disponibles, sin dependencia de productos propiedad de terceros.
- La instalación y configuración del prototipo es más sencilla.
- Se abandonan las pruebas referentes a generación de código, puesto que ya no tenía interés continuar con ellas.
- Verificación del sistema, similar a la de Itacio-SEDA.
- Este sistema implementa el modelo Itacio con mayor fidelidad que Itacio-SEDA, que era sólo un primer paso.

**Limitaciones más notables**

- Itacio-XJ se basa en una versión del modelo Itacio [CLC01a] que aún no contemplaba la conexión de una fuente con más de un sumidero ni de un sumidero con más de una fuente (adolece de la misma limitación que Itacio-SEDA en este aspecto).
- Se abandona la edición gráfica, aunque se mantiene la representación gráfica.
- La edición mediante ficheros de texto resulta rudimentaria, aunque sea flexible.
- Sólo puede realizarse la edición de los ficheros de texto trabajando directamente en el ordenador servidor; a través de Internet sólo pueden realizarse consultas y validaciones.

**Conclusiones**

El segundo prototipo de Itacio, Itacio-XJ, también cumplió plenamente su misión. Permitió ganar confianza sobre la viabilidad de la idea, y pudo ser aplicado a algunos casos de estudio; por ejemplo, se utilizó este prototipo para realizar pruebas sobre la verificación de contratos de reutilización [CLC01c] o sobre el diagnóstico remoto de equipos Windows [Ce01]. La aplicación de Itacio a problemas para los que no se había diseñado

específicamente demostró la hipótesis de que el modelo era muy flexible y aplicable a diferentes problemas del desarrollo, como mantenimiento y explotación.

No obstante, el modelo seguía requiriendo una reformulación para dar soporte a la conexión de una fuente con varios sumideros. Además, parecía apropiado abordar una implementación basada en base de datos, lo que sin duda constituía una técnica más cercana a lo que podría ser un sistema basado en Itacio para su explotación comercial. Uniendo ambos objetivos, se abordó el desarrollo de un tercer prototipo.

### 5.1.3. Tercer prototipo: Itacio-XDB

**Antecedentes**

Como ya se ha indicado, se trataba de reflejar en un prototipo el nuevo algoritmo de generación de bases de conocimientos que permitía la conexión de una fuente a varios sumideros, cosa que mejora drásticamente la operatividad del modelo. Asimismo, se pretendía verificar en la práctica que, nuevamente con medios humanos y técnicos muy limitados, se podía desarrollar un prototipo de Itacio, esta vez apoyado en el uso de una base de datos convencional como almacén de las descripciones de los componentes y los sistemas; se pretendía también comprobar que efectivamente una base de datos convencional podía encajar en el esquema general del modelo.

Se abordó el desarrollo de Itacio-XDB en los primeros días de agosto de 2001, y una primera versión estuvo disponible a finales de ese mismo mes. El nombre Itacio-XDB hace referencia a las tecnologías que siguen presentes (XML / XSL) y al uso de una base de datos como soporte de la información sobre componentes y sistemas (de ahí DB).

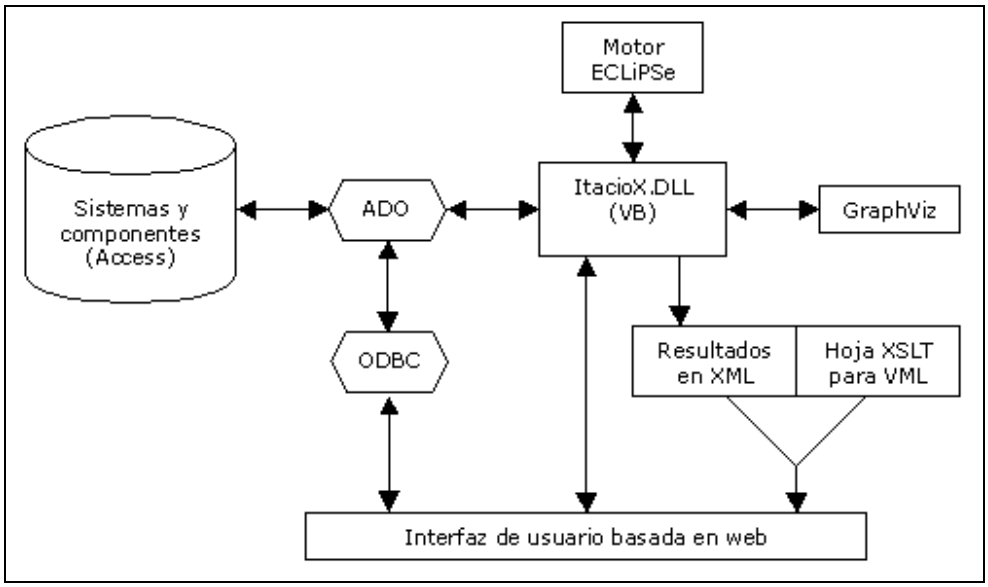



*Ilustración 16. Estructura del prototipo Itacio-XDB.*

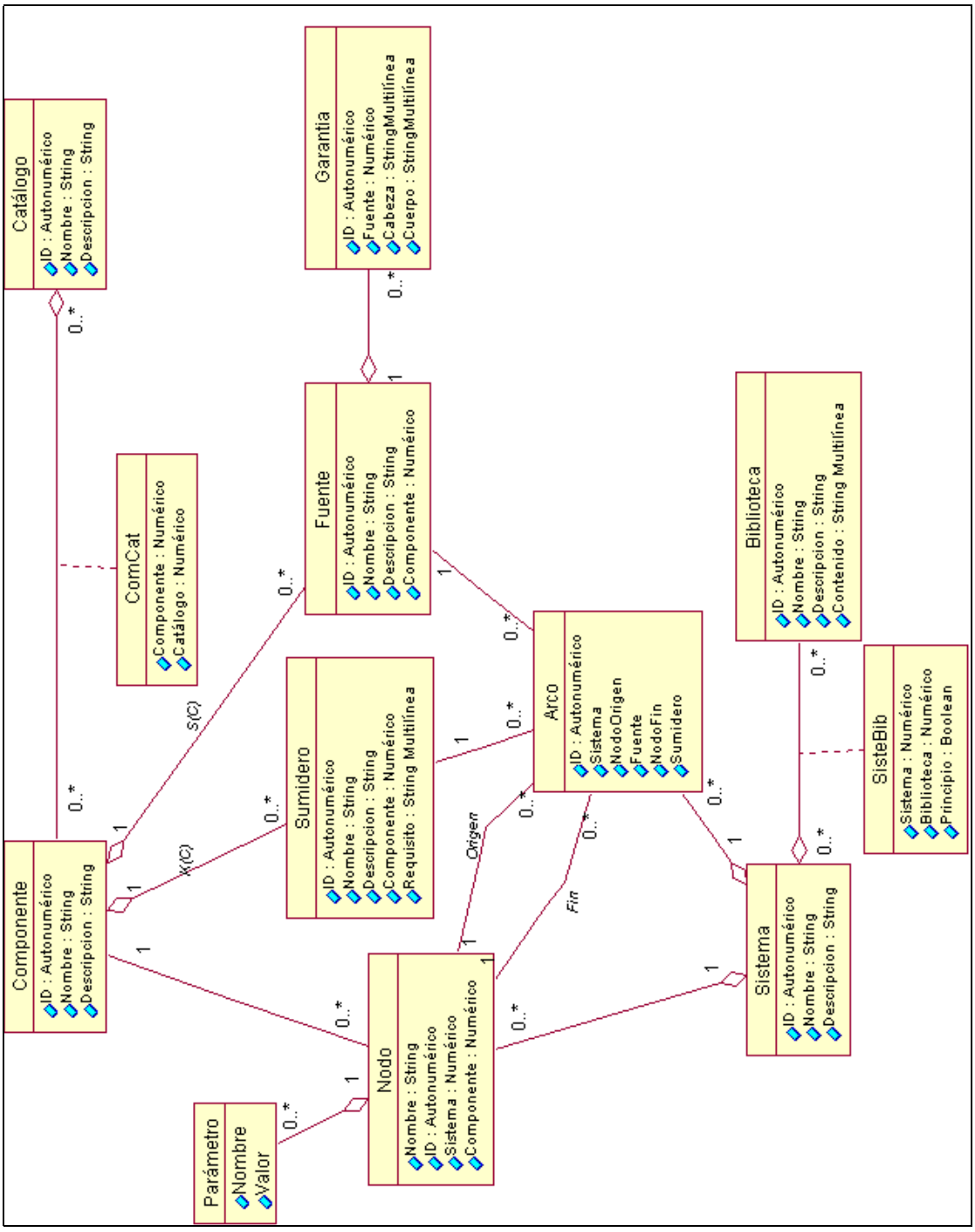



*Ilustración 17. El modelo de datos de Itacio-XDB.*

**Estructura del prototipo**

El núcleo de Itacio-XDB se apoya, como ya se ha dicho, en una base de datos (Ilustración 16). En este caso, se ha optado por Microsoft Access 2000, precisamente con la intención de probar que no es necesaria una tecnología específica como base para el prototipo. Una

estructura de tablas convencional (Ilustración 17) contiene la información sobre componentes, expresiones restrictivas, instancias de componentes (aquí denominadas "nodos"), sistemas, etc.La interfaz de usuario sigue siendo una interfaz ligera desarrollada en web. En este caso, y puesto que la plataforma de desarrollo es Windows, se ha optado de nuevo por el Microsoft Personal Web Server como servidor web, lo que permite el uso de ASP (*Active Server Pages*) [KSW99, Mi97] para crear una interfaz de usuario que acceda a la base de datos a través del web.

La estructura de páginas se creó de forma que se minimizase el coste de desarrollo; se crearon páginas genéricas, que suministrándoles los parámetros adecuados pueden dar altas, bajas y modificaciones en distintas tablas (Ilustración 18). Itacio-XDB puede actuar sobre distintas bases de datos (para que el usuario pueda organizar sus experimentos más fácilmente), pero existe una base de datos permanente, itacioUI.MDB, que contiene la información sobre las tablas, listados, etc.; así, muchas ampliaciones y cambios pueden realizarse alterando la información de itacioUI.MDB sin necesidad de modificar las páginas ASP. Gran parte de la interfaz de usuario es, pues, dinámica.

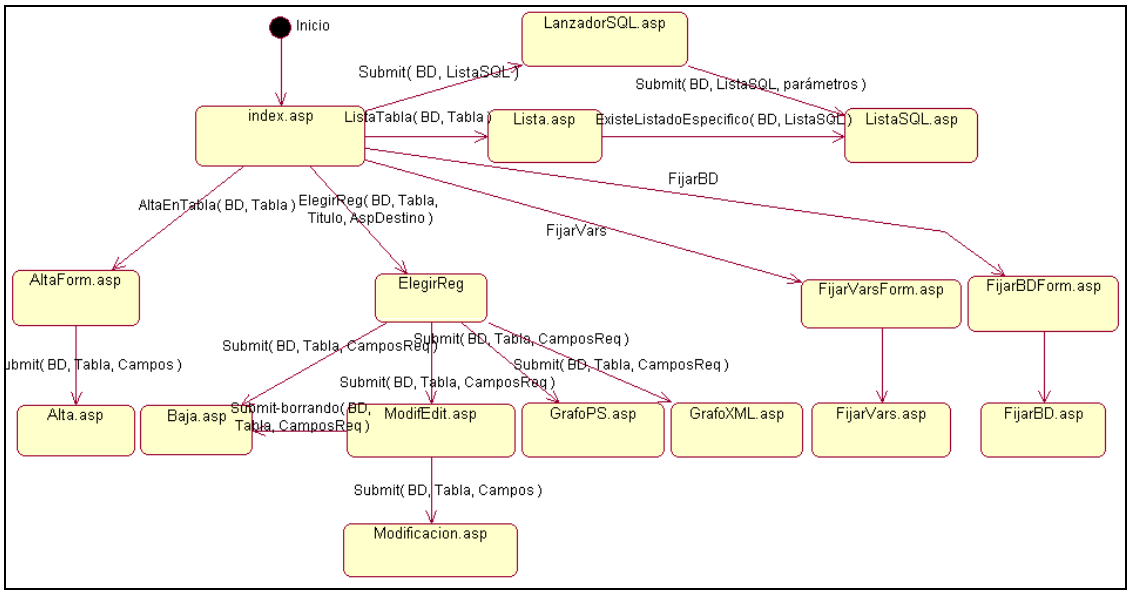



*Ilustración 18. Estructura de páginas de la interfaz de usuario de Itacio-XDB.*

Las páginas ASP ofrecen algunas facilidades de edición mediante formularios. Por ejemplo, una ventaja de Itacio-XDB frente a los prototipos anteriores es que puede ejecutarse realmente a distancia, con todas las consecuencias (incluyendo la modificación de las bases de datos), sin necesidad de estar utilizando el ordenador servidor (Itacio-XJ permitía realizar consultas desde cualquier equipo, pero la edición de textos sólo podía realizarse directamente en el equipo servidor). Al igual que en Itacio-XJ se ha renunciado a una edición gráfica de la estructura de los sistemas, y el realizar estas tareas mediante formularios resulta relativamente engorroso; pero ya se ha dicho que la alta productividad del prototipo no reviste especial importancia, y que era prioritario desarrollar el prototipo en un tiempo reducido.

Lógicamente, el funcionamiento del prototipo no se acaba con la edición de la información contenida en la base de datos. Debe ofrecer también soporte para los procesos de

generación de la base de conocimientos, generación de representación visual de grafos, validación, etc. Para programar todo esto de manera conveniente, se ha creado un componente ActiveX en Visual Basic, ItacioX, cuyos métodos realizan precisamente estos cometidos. Puesto que ASP puede cargar (en el servidor) un objeto ActiveX e invocar a sus métodos, resulta fácil que las páginas ASP recurran a este componente para los procesos más complejos, que pueden así programarse en un lenguaje más apropiado.

En este caso, una primera tarea de ItacioX es la representación gráfica de los sistemas, en forma de grafo. En Itacio-XJ se utilizaba la biblioteca GFC en Java; aunque su uso fue relativamente satisfactorio, Itacio-XDB se apoya en una alternativa mejor, el sistema GraphViz y concretamente su programa *dot* [KK93]. Este programa, dada información sobre la topología de un grafo, es capaz de ubicar los nodos del mismo de forma "adecuada", y generar esa representación tanto en formato PostScript como en un formato propio que informa de las posiciones en las que debe ubicarse cada elemento. ItacioX es capaz de leer este segundo formato, generando el XML correspondiente que permite mostrar la página gráficamente en el navegador de Internet mediante VML, como ocurría en el prototipo Itacio-XJ. Por tanto, Itacio-XDB es capaz de ofrecer gráficamente los resultados de una validación, tanto en VML como en PostScript. En el caso de VML, además, el diagrama será interactivo al igual que ocurría en Itacio-XJ, y al pulsar sobre las conexiones incorrectas se abrirá una ventana que indicará la razón del fallo.

La segunda tarea importante de ItacioX es la generación de la base de conocimientos de un sistema y la interacción con un motor de inferencia para deducir qué conexiones son incorrectas. Esto se consigue nuevamente a través del componente InterECL, que sirve de interfaz con ECLiPSe. Hay que hacer notar, además, que ItacioX genera la base de conocimientos de acuerdo con el procedimiento descrito en esta tesis (capítulo 4, pág. 99) y por tanto ya contempla conexiones múltiples, por ejemplo de una fuente con varios sumideros, algo que no permitían los prototipos precedentes y que motivó, entre otras cosas, el desarrollo de Itacio-XDB.

La interfaz de usuario de Itacio-XDB, dejando aparte la dificultad de la edición no-gráfica de los sistemas, es mucho más completa y manejable que la de Itacio-XJ.

Como indicación aproximada del tamaño de este prototipo, el módulo ItacioX incluye unas 1780 líneas de código Visual Basic (incluyendo la funcionalidad específica descrita en el apartado "Extensión de Itacio-XDB específica para WaveX" del capítulo 5.2.4). La interfaz de usuario consta de algo más de 1400 líneas de código ASP, distribuidas en 24 archivos. Todas estas cifras incluyen las líneas de comentarios. Puede verse que, al igual que en el prototipo Itacio-XJ, el tamaño del sistema es ciertamente reducido.

**Capacidades del prototipo**

- Como ya se ha dicho, Itacio-XDB responde a la idea de interfaz de usuario ligera y el 100% de su funcionalidad puede ejecutarse a distancia, sin más requisito que un navegador web capaz de mostrar VML y/o con soporte para ver ficheros PostScript.
- Soporta el uso de varias bases de datos, lo que facilita la organización de los casos de estudio.

- El desarrollo y ampliación de Itacio-XDB es muy sencillo, puesto que buena parte de la estructura de la interfaz de usuario se encuentra descrita *declarativamente* en una base de datos específica, itacioIU.MDB.
- Es el primer prototipo que da soporte al modelo final de Itacio, que incluye la posibilidad de conexiones múltiples entre fuentes y sumideros.

**Limitaciones más notables**

- Itacio-XDB sigue adoleciendo de la falta de un editor gráfico para describir los sistemas, aunque no era un objetivo de su construcción.
- Como prototipo que es, presenta diversos defectos de funcionamiento y la agilidad de uso es muy mejorable.

**Conclusiones**

En el proceso de verificar la viabilidad técnica del modelo Itacio, este tercer subproyecto permitió demostrar que en un tiempo de aproximadamente un mes y empleando exclusivamente tecnologías fácilmente accesibles es posible tener funcionando un prototipo de funcionalidad significativa. Parece evidente que las necesidades adicionales que plantearía un sistema destinado a explotación (como el editor gráfico mencionado) son, en principio, plenamente resolubles mediante técnicas de desarrollo convencionales.

El no haber encontrado ningún obstáculo reseñable en su construcción hace pensar que las previsiones iniciales son correctas. Además, este tercer prototipo ha permitido experimentar con el modelo Itacio en nuevos campos de aplicación; por ejemplo, en un sistema de componentes "convencional" [CLC01d], que figura entre los últimos experimentos realizados y que como era de esperar ha respondido de manera normal a la aplicación del modelo (ya que primero se abordaron las aplicaciones menos previsibles y más novedosas y el modelo respondió de manera correcta).

### 5.1.4. Conclusiones sobre los prototipos

Valorando de manera global la actividad experimental desarrollada en esta tesis, cabe decir que el modelo Itacio ha respondido perfectamente a su aplicación en ámbitos y problemas diferentes, problemas que en absoluto se habían previsto cuando se formuló el modelo. La hipótesis de que su flexibilidad permitiría aplicarlo a cualquier situación en la que se pudiese realizar una *instanciación* o adecuación entre los elementos reales y los objetos del modelo se vio, pues, apoyada en la práctica. Volvamos a los objetivos planteados para este trabajo experimental:

**La verificación del software debe ser estática**. Como se explicó entonces, por la propia definición del método parecía obvio que se cumplía este requisito. No obstante, la implementación de estos prototipos permite demostrar sin lugar a dudas que la verificación del sistema es, en efecto, estática: algunos de los sistemas descritos y verificados en Itacio no llegaron a construirse, y se trabajó en ellos sólo en un nivel de diseño.

**La verificación del software debe ser automática**. La implementación de los algoritmos de verificación en un sistema informático plenamente funcional prueba también que no parece haber dificultades de automatización insoslayables en ningún elemento del modelo.

Aunque su uso pueda ser más o menos eficiente, los prototipos aquí desarrollados realizan su labor de forma totalmente automática.

**Susceptible de ser implementado mediante técnicas conocidas y viables**. Este es uno de los puntos que más se han beneficiado de la realización de los prototipos. Sobre lo conocido de las técnicas, es evidente que XML, Java, Microsoft Access, Visual Basic o ASP no se pueden calificar de alta tecnología por lo que se refiere a su disponibilidad y difusión; son técnicas de desarrollo muy extendidas y que las organizaciones de desarrollo aplican de manera rutinaria. Sobre la viabilidad, puede decirse algo similar; existen multitud de profesionales preparados para su uso. La conclusión de esto es que en principio es posible desarrollar un sistema de producción basado en el modelo Itacio utilizando solamente recursos técnicos convencionales.

Sobre la viabilidad, debe tenerse en cuenta que, además de haber empleado sólo técnicas conocidas, los recursos de desarrollo utilizados eran extremadamente limitados, y se ha hecho así de manera deliberada. Sólo ha intervenido en el desarrollo una persona, que es el autor de esta tesis. Puede aducirse que esto es una ventaja, puesto que el diseñador / programador no encontrará dificultades para entender su propio análisis; pero también es cierto que este programador desconocía en gran medida las técnicas involucradas, lo que compensa notablemente la ventaja mencionada. En concreto, el desarrollo de Itacio-SEDA sirvió de aprendizaje de ECLiPSe, entre otras cosas. El desarrollo de Itacio-XJ constituyó el **primer** contacto del autor con XSL y VML, y una de sus primeras implementaciones en Java. El desarrollo de Itacio-XBD fue el **primer** contacto con ASP; es decir, ese mes y medio de desarrollo incluye el aprendizaje desde cero de la tecnología ASP. El hecho de que los prototipos se hayan podido construir en un tiempo corto, con recursos humanos muy limitados, en proyectos altamente tentativos y en los que el personal involucrado no estaba altamente cualificado (puesto que buena parte del tiempo se dedicó a formación y entrenamiento) permite afirmar con ciertas garantías que Itacio puede ser implementado en una aplicación plenamente funcional sin grandes problemas técnicos.

**Susceptible de ser comprendido y aplicado en la práctica con relativa facilidad por personal de desarrollo**. Ya se ha mencionó (pág. 114) que este punto era particularmente difícil de demostrar, y que no cabía más que una aproximación práctica y unas previsiones en el plano teórico. La implementación del modelo en los prototipos demuestra que la propia herramienta puede construirse; no permite demostrar que el *uso* de esa herramienta resulta sencillo. Sin embargo, esto puede inferirse a partir de los experimentos realizados (capítulo 5.2), que sí son experimentos de *uso* de la herramienta y del modelo conceptual en el que se basa. Asimismo, la facilidad con que se ha desarrollado la herramienta hace pensar que cuando menos las tecnologías implicadas son manejables (caso de ECLiPSe), y esta ventaja sería apreciable también en el uso.

**Flexible y susceptible de ser aplicado en diferentes ámbitos, no demasiado específico**. Nuevamente, este aspecto se ha intentado verificar mediante el uso de los prototipos, no mediante su construcción. Bien es cierto que los diferentes prototipos se han implementado de maneras distintas y con técnicas distintas, cosa que también refuerza la idea de la generalidad del modelo.

## 5.2. Casos de aplicación

La construcción de los prototipos descritos en el punto anterior ha permitido, pues, reforzar la idea de que el modelo responde a los propósitos de verificación estática y automática, y también de que puede ser implementado mediante técnicas conocidas y viables a un coste razonable. Ello nos lleva a la conclusión de que construir herramientas basadas en Itacio es perfectamente factible.

Otro aspecto de la cuestión inicial es la generalidad y flexibilidad del modelo. La construcción de los prototipos puede dar alguna idea a este respecto, pero lo que puede aportar información más valiosa al respecto es el uso de esos prototipos para describir diferentes problemas. Esa es la tarea que se ha abordado utilizando los diversos prototipos, y cuyos resultados se detallan a continuación.

### 5.2.1. Microcomponentes

Un primer nivel en el que se han aplicado las ideas de Itacio fue el nivel de *microcomponentes*.

Es muy cierto que los problemas de fondo más graves en el desarrollo de un sistema informático provienen de ámbitos de nivel de abstracción muy superior, especialmente la recogida y gestión de requisitos, el análisis y el diseño. Estos problemas, además, pueden llevar a la obtención de sistemas de utilidad cuestionable (por no adaptarse a las necesidades del usuario), de bajo rendimiento o, y este es quizás el problema más frecuente y grave, de muy alto coste de mantenimiento.

Con frecuencia, este tipo de defectos de ámbito amplio (aliados con malas prácticas de estimación, de gestión de proyectos, de gestión de riesgos y de estrategia de desarrollo) son los causantes de la cancelación de los proyectos. Pero cuando se habla de defectos en la construcción de software, con frecuencia se subestiman los defectos cuyo origen se encuentra en el nivel de lo que en este documento se denomina *microcomponentes*.

Parece lógica la postura de centrar el estudio metodológico en los ámbitos descritos arriba como de mayor impacto en el desarrollo de un proyecto; la mayor preocupación de la ingeniería informática es proporcionar herramientas que permitan realizar mejor tareas de un nivel de abstracción relativamente alto. Las tareas de bajo nivel de abstracción, es decir, la *programación* propiamente dicha, quedan en manos del "factor humano".

Pro supuesto, hay muchos aspectos metodológicos que afectan a la programación. La redacción de código puede estar sujeta a estándares de calidad. Pueden estar en vigor normas de nomenclatura y formato, o directrices sobre las construcciones a utilizar en el lenguaje (los *idioms* particulares). Pueden aplicarse también criterios cuantitativos a través de diversas métricas, detectar módulos conflictivos o de complejidad ciclomática demasiado elevada. Y en el mejor de los casos, el sistema de pruebas imperante exigirá unas pruebas unitarias y de regresión automatizadas y documentadas. Por supuesto, un capítulo importante en la redacción de código es el uso de herramientas elegidas basándose en criterios de calidad, con un sistema de comprobación estática de tipos si así se considera conveniente, con un entorno de desarrollo que fomente la productividad y la detección temprana de errores.

Pero a pesar de todo esto, que desde luego contribuye a evitar errores, finalmente el programador se encuentra solo en fases significativas del desarrollo. No por su influencia

conceptual sobre el sistema, que lógicamente será reducida si se atiene a las especificaciones de diseño, sino por su posible influencia en la tasa de defectos finales. Aun dentro de los márgenes aparentemente estrechos que existirían en una situación como la descrita en el párrafo precedente, hay lugar suficiente para que se produzcan defectos, que en muchos casos pueden quedar ocultos hasta etapas avanzadas.

Estos defectos pueden resultar problemáticos: pueden manifestarse sólo en situaciones muy concretas no cubiertas por los casos de prueba, se trata de defectos de "pequeño tamaño" (difíciles de detectar), y su efecto sobre el análisis o diseño es nulo pero su efecto en ejecución puede ser determinante.

Parece atractiva la idea de aplicar un modelo de verificación a este nivel, que nos acerque a la noción de *software correcto por construcción*. Puesto que los diversos métodos formales tienen las ventajas e inconvenientes ya mencionados, este es un campo candidato a la aplicación de Itacio. La única particularidad es que los componentes involucrados son de muy pequeño tamaño, en el nivel del propio código fuente (de ahí el término *microcomponentes*).

La idea del software correcto por construcción, que es una de las ideas motoras de este proyecto de investigación, ha sido bien resumida por Wallnau y Plakosh en [WP00]:

> *Lo que estamos sugiriendo es algo análogo a lo que motivó la programación estructurada. Si analizar programas de ordenador con estructuras GOTO arbitrarias es problemático, la solución no es mejorar las herramientas de análisis (eso podría ser útil), sino estructurar los programas de modo que sean analizables (esto será útil).*

Refiriéndose en concreto a las ventajas de una estructuración basada en componentes, continúan:

> *Análogamente, los marcos de componentes pueden imponer diversas restricciones arquitectónicas que se expresan en un modelo de componentes. Bien diseñado (es decir, estructurado), un modelo de componentes puede hacer que los sistemas sean más fáciles de analizar con respecto a una o más propiedades interesantes.*

En nuestro caso concreto, se han realizado pruebas sobre un lenguaje imperativo tradicional como es C, considerando componentes a las diversas funciones de biblioteca, operadores del lenguaje, etc. Esto ha permitido expresar "en algún sitio" que la raíz cuadrada, por ejemplo, debe recibir valores que se garantice que son positivos, y que el denominador de una división nunca puede ser 0.

La conclusión con estas experiencias es que para llevar a la práctica este enfoque es necesario relacionar directamente el modelo de microcomponentes con un modelo de generación de código. En nuestro caso se ha preparado (en el prototipo Itacio-SEDA; véase capítulo 5.1.1) una implementación elemental de esta idea. Ello permitía diseñar gráficamente sencillos programas de cálculo matemático (con operaciones básicas como las mencionadas) ensamblando componentes. Estos componentes, además de información sobre restricciones, llevaban asociada información para genera código, y así los sistemas construidos podían transformarse en programas C, que se sabía que eran "correctos por construcción" (y siempre y cuando, claro está, las expresiones restrictivas de los operadores involucrados fueran correctas). En la Tabla 5 pueden verse algunos microcomponentes de ejemplo.

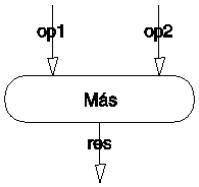
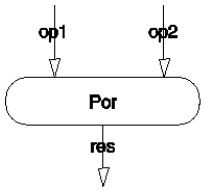
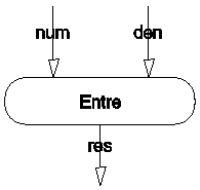
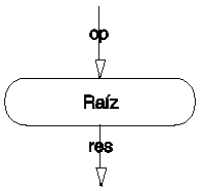
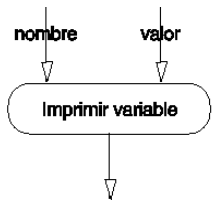
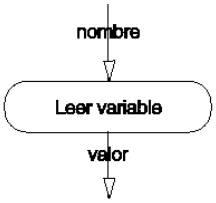
| Componente | Descripción | Expresiones restrictivas |
|---|---|---|
|  | Realiza la suma de los dos operandos. | `mayor_o_igual_que($res$, 0) :-`<br>`mayor_o_igual_que($op1$, 0),`<br>`  mayor_o_igual_que($op2$, 0).`<br>`mayor_que($res$, 0) :- mayor_que($op1$, 0),`<br>`  mayor_o_igual_que($op2$, 0).`<br>`mayor_que($res$, 0) :- mayor_o_igual_que($op1$, 0),`<br>`  mayor_que($op2$, 0).` |
|  | Realiza el producto de los dos operandos. | `mayor_o_igual_que($res$, 0) :- igual_que($op1$, $op2$).` |
|  | Realiza la división del numerador por el denominador. | `valor_aceptable($den$) :- !,`<br>`  write('Comprobar denominador <> 0...'), nl,`<br>`  distinto_de(den, 0),`<br>`  write('Lo es.'), nl.`<br>`mayor_o_igual_que($res$, 0) :-`<br>`mayor_o_igual_que($num$, 0),`<br>`  mayor_que($den$, 0).`<br>`mayor_o_igual_que($res$, 0) :-`<br>`menor_o_igual_que($num$, 0),`<br>`  menor_que($den$, 0).`<br>`[otras que no se incluyen por razones de espacio]` |
|  | Calcula la raíz cuadrada real y positiva del operando. | `valor_aceptable($op$)  :- !,`<br>`  write('Comprobar si es positivo...'), nl,`<br>`  es_positivo($op$),`<br>`  write('Lo es.'), nl.`<br><br>`es_positivo($res$).`<br>`menor_o_igual_que($res$, $op$) :-`<br>`mayor_o_igual_que($op$, 1).`<br>`mayor_o_igual_que($res$, $op$) :- menor_que($op$, 1).` |
|  | Muestra por pantalla el valor de una variable. | |
|  | Pide por teclado al usuario el valor de una variable. | |

*Tabla 5. Algunos microcomponentes en Itacio-SEDA.*

Estos experimentos no llegaron mucho más allá que a la descripción de estos programas elementales de cálculo. Lo contrario habría requerido el desarrollo de un modelo de generación de código mucho más ambicioso, cosa que en absoluto era un objetivo de la tesis. El resultado de estos experimentos fue primero comprobar que conceptualmente nada impedía la aplicación del modelo Itacio a los microcomponentes, proporcionando una alternativa para incorporar conocimiento a este nivel, y que había dificultades de orden práctico relacionadas con el desarrollo de un entorno productivo para la creación de código basado en microcomponentes y la generación de código a partir de los mismos.

## 5.2.2. Verificación de contratos de reutilización

Según el modelo de proceso presentado anteriormente, el uso de Itacio se basa en *instanciar* el modelo, adaptando las definiciones teóricas de Itacio a las entidades reales que se manejen en cada caso. Ajustando los conceptos del modelo y del dominio del problema, se puede aplicar Itacio a diferentes ámbitos.

Un ejemplo peculiar de esto es la verificación de la **evolución** de un sistema. En general, cuando se habla de componentes puede haber ciertas divergencias en lo que cada cual entiende por componente, pero casi siempre se hace referencia a entidades estructurales o morfológicas de un sistema. Sin embargo, en este caso se aplica el modelo de componentes a la evolución temporal de un sistema, a los efectos de las sucesivas modificaciones sobre las restricciones del sistema. La estructura de colaboraciones de un sistema se modela como un componente, y las distintas modificaciones a dicha estructura se consideran a su vez componentes. De este modo, las conexiones entre componentes describen líneas temporales, y no estructurales.

El modelado de la evolución del sistema se basa en los contratos de reutilización de Lucas et al, mencionados en el capítulo 2. Aunque allí se ofrecía una visión general del enfoque que el grupo PTL-VUB daba al concepto de "contrato", en este punto es necesaria una descripción más detallada como introducción del trabajo realizado con Itacio.

**Contratos de reutilización**

La teoría de contratos que se maneja en este caso es la de Carine Lucas [Lu97]. Podemos resumir aquí las siguientes definiciones:

**Def. XVIII:** Un **contrato de reutilización** es un conjunto de participantes, cada uno de los cuales tiene un nombre (único dentro del contrato de reutilización), una cláusula de dependencia (*acquaintance clause*) y una interfaz.

**Def. XIX**: Una **cláusula de relación** es la indicación de que un participante de un contrato tiene relación con otro participante. Si un participante "sabe de la existencia" de otro participante, el primero tendrá una cláusula de relación en la que se mencionará al segundo. Además, el primero se referirá al segundo mediante un *nombre de relación*, una especie de alias recogido en la cláusula.

**Def. XX**: La **interfaz** de un participante es básicamente un conjunto de descripciones de operaciones (teniendo el término *operaciones* un significado similar al de los *métodos* de orientación a objetos). Cada descripción contiene un nombre de operación y una lista de las operaciones a las que esta invoca; las operaciones invocadas aparecen calificadas con el nombre de relación.

Con estos elementos es posible modelar ciertos aspectos de la colaboración entre un conjunto de participantes. Un contrato de reutilización contiene información sobre los participantes, las operaciones que implementan, los participantes que necesitan conocer para ello, y las dependencias entre operaciones.

Es posible construir un contrato de reutilización inconsistente; por ejemplo, las operaciones pueden hacer referencia a operaciones o participantes inexistentes. Para evitar esto, se definen criterios de corrección; un contrato se dice bien formado si satisface tres cláusulas de buena formación, llamadas WF1, WF2 y WF3.:

- **WF1** requiere que las cláusulas de relación hagan referencia a participantes existentes.
- **WF2** requiere que las dependencias entre operaciones hagan referencia a relaciones válidas.
- **WF3** requiere que la operación invocada exista en el participante al que se hace referencia a través de la cláusula de relación.

El trabajo original de Lucas no establece más cláusulas porque su notación evita implícitamente cualquier otro error (debido a la propia sintaxis en la que se expresan los contratos, si esta se respeta no se cometerán otros errores). Pero si se utiliza una notación diferente, pueden ser necesarias cláusulas de buena formación adicionales. En nuestro caso, no se utiliza la notación de Lucas para describir los contratos, sino que estos se describen mediante las expresiones restrictivas de Itacio, y por tanto mediante cláusulas Horn; al ser esta notación mucho más flexible que la de Lucas, es posible cometer otro tipo de inconsistencias, que hay que verificar con reglas adicionales.

El trabajo de Lucas parte de la definición del contrato de reutilización, pero llega mucho más lejos: incluye un estudio de las modificaciones que se pueden realizar sobre un contrato. Estas posibles modificaciones toman la forma de **operadores** (evítese la confusión entre el término *operador* y *operación*, ya utilizado previamente en la Def. XX). Los operadores se agrupan por dos criterios: por un lado, su ámbito, y por otro lado, el tipo de modificación que realizan.

Agrupando los operadores según el ámbito, tenemos:

- Modificaciones de **participante**, que afectan a las operaciones de los participantes. Es decir, el ámbito de la modificación es el interior de un solo participante.
- Modificaciones de **contexto**, que afectan a la existencia de los participantes o a la relación entre ellos. Es decir, el ámbito de la modificación alcanza más allá de un solo participante.

Nótese que en principio la terminología puede parecer algo confusa. Podría pensarse que la eliminación de un participante es una "modificación de participante", pero recuérdese que el nombre de la categoría se refiere al ámbito de la modificación, y eliminar un participante completo tiene su efecto en el grupo de participantes, no es una modificación interna al mismo.

La otra agrupación posible es, pues, basándose en el tipo de modificación:

- Modificaciones **de extensión**, lo que implica añadir elementos.
- Modificaciones **de cancelación**, cuando se eliminan elementos.
- Modificaciones **de refinamiento**, cuando se añaden dependencias entre elementos.
- Modificaciones **de generalización**[4], cuando se eliminan dependencias entre elementos.

[4] El término *generalización* en este caso es una interpretación del inglés *coarsening*, que literalmente significa "volver más basto o tosco". A falta de un buen antónimo de "refinar", se ha optado por aludir a la idea de que el *coarsening* elimina detalles y puede considerarse una generalización.

Si se combinan los dos posibles criterios, ortogonales entre sí, se obtienen ocho combinaciones, que son los ocho operadores básicos o canónicos (nótese la posible confusión terminológica a la que aludíamos antes):

- **Extensión de participantes**. Consiste en añadir operaciones a un participante ya existente.
- **Extensión de contexto**. Consiste en añadir participantes a un contrato.
- **Cancelación de participantes**. Consiste en eliminar operaciones de un participante.
- **Cancelación de contexto**. Consiste en eliminar participantes de un contrato.
- **Refinamiento de participantes**. Consiste en añadir dependencias (es decir, llamadas) entre las operaciones de los participantes.
- **Refinamiento de contexto**. Consiste en añadir cláusulas de relación entre los participantes.
- **Generalización de participantes**. Consiste en eliminar dependencias (llamadas) entre las operaciones de los participantes.
- **Generalización de contexto**. Consiste en eliminar dependencias entre los participantes, es decir, cláusulas de relación.

Estos operadores están definidos formalmente en [Lu97]. En primer lugar se define claramente la estructura de sus respectivos parámetros. En términos generales, un operador recibe como primer parámetro un **contrato**, y como segundo parámetro un **modificador**. El modificador es una lista de participantes, operaciones, etc.; con ambos parámetros, el operador genera un nuevo contrato modificado de manera conveniente.

Existen consideraciones semánticas sobre estos parámetros. No basta con que el contrato y el modificador sean, por separado, morfológicamente correctos: no siempre es posible aplicar un modificador sobre cierto contrato. Como ejemplo obvio, si se intenta realizar una cancelación de contexto, eliminando un participante, y el nombre de participante a eliminar no existe en el contrato original, no será posible tal cancelación. Existen criterios similares para los diferentes operadores, lo que da lugar al concepto de *aplicabilidad* de un modificador y operador sobre cierto contrato.

Hay que hacer notar que cada modificador puede incluir participantes, cláusulas de relación, operaciones, etc. y esto provoca que, en gran medida, los modificadores tengan un aspecto similar a contratos o a fragmentos de contratos. Por tanto, es lógico adoptar para los modificadores la misma notación que para los contratos, lo que facilita la comprensión del esquema propuesto y la implementación de los prototipos.

Los ocho operadores de reutilización mencionados más arriba son **atómicos**, en el sentido de que cada operador recoge una de las modificaciones básicas que se pueden aplicar a un contrato. También son plenamente **modulares**; cada uno de esos operadores se puede aplicar y verificar de manera independiente. Cabe también destacar que, como se demuestra en [Lu97], si se dispone de un contrato de reutilización bien formado (es decir, que cumple las reglas WF1, WF2 y WF3), se toma un modificador y un operador, y dicho modificador es *aplicable* al contrato mediante el operador elegido, el contrato resultante estará a su vez bien formado. Estas características –atomicidad, modularidad y coherencia- son positivas,

pero el uso de estos operadores básicos presenta también sus problemas de orden práctico: cada operador representa un cambio muy limitado sobre un contrato.

Lo habitual es que cualquier modificación práctica implique la aplicación de varios operadores; por esta razón, Lucas desarrolló el concepto de **operadores combinados**. Estos operadores son simplemente compuestos de los operadores atómicos. En nuestro caso, no se han utilizado operadores combinados, porque los ocho operadores canónicos, aunque poco ágiles en tiempo de explotación, son suficientes para nuestros propósitos y en cualquier caso los operadores combinados pueden reducirse a operadores canónicos.

Cuando se aplican varias modificaciones de manera sucesiva a un contrato, pueden darse errores o inconsistencias a medida que el sistema evoluciona. Para afrontar este problema, Lucas evalúa también las posibles combinaciones entre operadores y propone diferentes reglas o *recetas* para evitar las combinaciones peligrosas, reglas desarrolladas específicamente como resultado de su estudio. Sin embargo, el uso de Itacio permite detectar los problemas sin necesidad de aplicar estas reglas específicas; el propio sistema de verificación de Itacio comprueba de manera implícita que el resultado de una cadena de modificaciones es correcto.

**Los contratos de reutilización considerados como componentes de Itacio**

Como ya se ha dicho, los contratos de reutilización son básicamente descripciones de la interacción entre varios participantes. Recogen información sobre las operaciones ofrecidas por cada participante, las relaciones de conocimiento mutuo entre los participantes (mediante las denominadas *cláusulas de relación*), y las operaciones invocadas por cada operación de un participante, ya sea dentro del mismo participante o a través de la cláusula de relación si necesita invocar operaciones de otros participantes.

Los participantes y sus operaciones son similares, en gran medida, a los objetos y sus interfaces. Un enfoque obvio sería identificar un componente con un objeto, y por tanto con un participante del contrato. La identificación entre componente y objeto es una cuestión que ya se ha tratado con frecuencia [Ha00, SW99 pág. 390, Ko00 pág. 33]. Itacio, además, podría perfectamente aplicarse bajo tal interpretación, que de hecho se encontraba en la motivación inicial de este trabajo. Pero no es este el enfoque que se aplica en este caso.

Aquí no se está aplicando el modelo de componentes de Itacio para verificar las conexiones estructurales entre un objeto y otro, sino para verificar la *evolución* de un sistema, de un conglomerado de participantes. La noción de componente se aplica, pues, en un nivel de abstracción más elevado. Cada contrato (*el contrato completo, incluyendo todos sus participantes*) será considerado como un solo componente. Del mismo modo, un modificador (es decir, un fragmento de contrato cuya misión es figurar como parámetro en un operador de reutilización) será considerado también un componente. Y a su vez, el operador en cuestión es otro componente.

Analizando esta interpretación, puede verse que un contrato de reutilización cumple los requisitos planteados en esta tesis para la noción de componente (véase 4.2): tiene una frontera bien delimitada y contiene un conjunto de expresiones restrictivas. La frontera en cuestión se considera que consta de una sola fuente: el nombre de contrato, que permite recuperar cualquier información sobre el mismo. Puede verse esta fuente como algo muy similar a un componente "valor entero"; este componente tendría una sola fuente que suministra un valor entero. En este caso, la única diferencia es que un valor "de tipo

contrato" es mucho más complejo que un valor "de tipo entero", y la información que contiene es mucho más rica, pero el papel general de ambos componentes sería esencialmente el mismo.

Ya se ha dicho que en esta aplicación particular del modelo no serían los contratos el único tipo de componente posible. Los operadores son también considerados componentes. Un operador tiene una sola fuente (el contrato que resulta de aplicar el operador) y dos sumideros: el contrato original y el contrato modificador. Nuevamente, puede resultar esclarecedor el ejemplo de un componente "suma de enteros": recibiría dos enteros y produciría un resultado también entero. Del mismo modo que en el caso de los componentes, el operador de contratos recibe y genera valores mucho más complejos que simples enteros, pero la estructura del operador y del componente de suma es idéntica.

**Contract: smplDrive**

| **PARTICIPANT**: smplDriver | **PARTICIPANT**: smplCar |
|---|---|
| **OPERATIONS**: | **OPERATIONS**: |
| ::go()<br>myCar::startEngine()<br>myCar::pushGasPedal()<br>::stop()<br>myCar::pushBrake()<br>myCar::stopEngine()<br>::goFaster()<br>myCar::pushGasPedal()<br>::goSlower()<br>myCar::pushBrake() | ::startEngine()<br>::stopEngine()<br>::pushBrake()<br>::pushGasPedal() |
| **ACQUAINTANCES**: | |
| smplDriver::myCar->smplCar | |

*Ilustración 19. Un contrato de ejemplo elemental.*

Dadas estas definiciones de componentes, es posible modelar la evolución de un sistema en el tiempo (a medida que el mantenimiento altera la colaboración entre objetos) como la aplicación de una cadena de operadores sobre los contratos. Por ejemplo, considérese el patrón de interacción entre un coche y su conductor. Este escenario podría describirse con un contrato de reutilización como el de la Ilustración 19. En este contrato, el participante `smplDriver` ofrece las operaciones `go`, `stop`, `goFaster` y `goSlower`. El participante `smplCar` ofrece las operaciones `startEngine`, `stopEngine`, `pushBrake` y `pushGasPedal`. Existe además una cláusula de relación (*acquaintance*) según la cual el participante `smplDriver` sabe de la existencia del participante `smplCar`, y se refiere a él mediante el alias `myCar`. Puede verse, además, que muchas operaciones de `smplDriver` aluden a las operaciones de `smplCar`; por ejemplo, para que smplDriver pueda implementar la operación `go`, debe invocar a las operaciones `startEngine` y `pushGasPedal` de `smplCar` (y en estas llamadas representa a dicho participante como `myCar`).

Toda esta información puede ser representada en un sistema de Programación Lógica con Restricciones mediante unas pocas sentencias declarativas (véase Ilustración 20). En el prototipo Itacio-XJ (véase capítulo 5.1) se ha diseñado una estructura de predicados capaz de representar la información de los contratos de reutilización de forma sencilla y legible. Además, se ha creado una biblioteca básica para manejar estas estructuras, obtener

información sobre las mismas y verificarlas; esta biblioteca sería el elemento ***L*** del modelo, mencionado en el capítulo 4.

```
Type=smplDrive
Sources=res

BEGIN_RESTRICTIONS
isContract($res$).

participant($res$, smplDriver).
participant($res$, smplCar).

acqRelationship($res$, smplDriver, myCar, smplCar).

operation($res$, smplDriver, go).
operation($res$, smplDriver, stop).
operation($res$, smplDriver, goFaster).
operation($res$, smplDriver, goSlower).

operation($res$, smplCar, startEngine).
operation($res$, smplCar, stopEngine).
operation($res$, smplCar, pushBrake).
operation($res$, smplCar, pushGasPedal).

operationInvocation($res$, smplDriver, go, myCar,
  startEngine).
operationInvocation($res$, smplDriver, go, myCar,
  pushGasPedal).
operationInvocation($res$, smplDriver, stop, myCar,
  pushBrake).
operationInvocation($res$, smplDriver, stop, myCar,
  stopEngine).
operationInvocation($res$, smplDriver, goFaster, myCar,
  pushGasPedal).
operationInvocation($res$, smplDriver, goSlower, myCar,
  pushBrake).

END_RESTRICTIONS
```

*Ilustración 20. Contrato de la Ilustración 19 expresado en el prototipo Itacio-XJ.*

Si fuese necesario modificar este contrato de modo que el coche tenga una radio y el conductor sea capaz de escuchar música, se podría construir un modificador de contrato que añadiese las operaciones necesarias a los participantes, como se describe en la Ilustración 21. Para poder combinar el contrato y el modificador y generar un nuevo contrato que reflejase la modificación realizada, habrá que utilizar un componente que represente al operador de contratos. En este caso, el operador en cuestión será una *extensión de participante*. Para que se pueda aplicar cualquier operador, tanto el contrato como el modificador deben cumplir ciertos criterios, que se engloban en lo que se ha llamado la *aplicabilidad* del modificador; son las expresiones restrictivas del operador las que se encargan de plantear estos requisitos.

| Contract: driveMusic | |
|---|---|
| **PARTICIPANT:** smplDriver<br>**OPERATIONS:**<br>::listenToMusic()<br>myCar::turnOnRadio()<br>::stopListeningToMusic()<br>myCar::turnOffRadio() | **PARTICIPANT:** smplCar<br>**OPERATIONS:**<br>::turnOnRadio()<br>::turnOffRadio() |
| **ACQUAINTANCES:**<br>smplDriver::myCar->smplCar | |

*Ilustración 21. Un modificador de contrato para añadir operaciones a los participantes.*

Por último, el contrato resultante de esta operación debe ser verificado. Entra en juego, pues, un nuevo componente que sólo ofrece un sumidero y ninguna fuente. El propósito del sumidero es simplemente la verificación del contrato que recibe en él. La única restricción que plantea este componente verificador es que el contrato que recibe esté bien formado.

Al combinar todos estos componentes en Itacio, construyendo el sistema que representa esta estructura, se obtiene un grafo que modela la evolución del contrato (Ilustración 22). En este caso, la combinación de componentes es correcta; en términos de teoría de contratos de reutilización, el contrato `smplDrive` está bien formado, y es *aplicable* una extensión de participante mediante `driveMusic`. Como era de esperar (y como demostró Lucas en [Lu97]) en estas condiciones la modificación no provoca ninguna incoherencia. Si se diera tal incoherencia, la conexión entre `addMusic` y `verification` aparecería señalada con un cuadrado rojo de mayor tamaño que los demás, y al pulsar sobre dicho cuadrado el usuario de Itacio-XJ vería una explicación sobre los motivos del fallo.

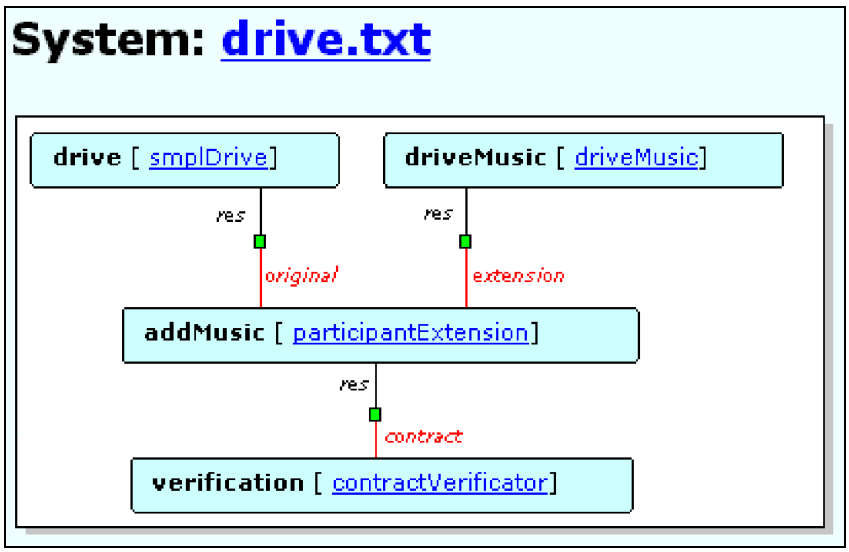



*Ilustración 22. Una modificación de contrato sencilla (captura de pantalla del prototipo Itacio-XJ)*

**Caso de estudio: guardar documentos en MFC**

Las Microsoft Foundation Classes o MFC [Po99] constituyen un gran conjunto de clases escritas en C++. No es sólo una biblioteca de clases, sino un marco de referencia completo para desarrollar aplicaciones destinadas a la familia de sistemas operativos Windows. Las herramientas de desarrollo de Microsoft generan *esqueletos* de aplicación en MFC, que el programador puede aprovechar como punto de partida. El desarrollo de aplicaciones nativas para Windows en C / C++ requiere una considerable cantidad de código de soporte e infraestructura (el aquí denominado "esqueleto"), por lo que el uso de MFC mejora notablemente la productividad. La contrapartida es la habitual cuando se utilizan marcos de desarrollo; la curva de aprendizaje es larga (ya se ha dicho que MFC es una biblioteca bastante grande) y además se depende de código de terceros, en este caso de código de Microsoft.

El ejemplo presentado aquí ha sido extraído de un caso real. Hay que aclarar por adelantado que la solución adoptada no era una solución deseable, pero en el momento en que se adoptó no parecía haber muchas más opciones. Independientemente de la posible crítica a este ejemplo concreto, lo importante es precisamente que sirve como ejemplo de situaciones similares.

La situación de partida era el desarrollo de un editor gráfico cliente / servidor. Uno de los requisitos para este sistema implicaba la modificación del esquema según el cual se grababan (*guardaban*) los documentos; cuando el usuario pulsase el botón *Guardar* del editor, era necesario realizar un procesamiento específico.

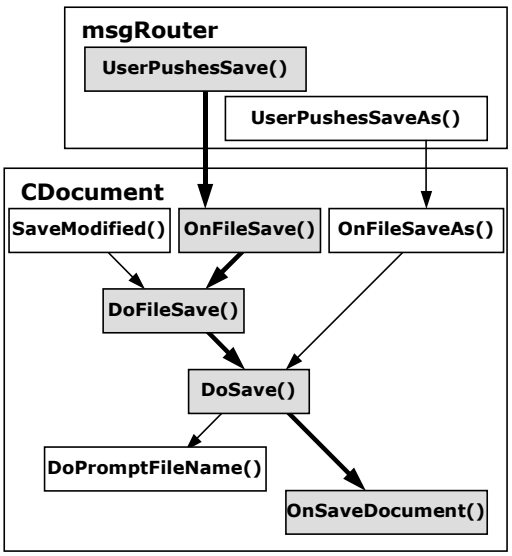



*Ilustración 23. Estructura original de MFC para guardar documentos.*

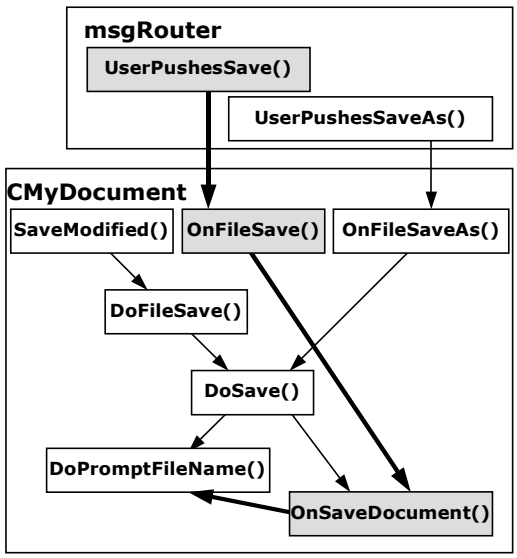



*Ilustración 24. Estructura de llamadas modificada en* `CMyDocument`

Como es habitual en los marcos de desarrollo de aplicaciones, el programador tenía que modificar el comportamiento de algunas clases, en este caso mediante herencia de implementación, apoyándose en el entorno que proporciona el código fuente de MFC. El esquema original de grabación de documentos se reproduce en la Ilustración 23. Los rectángulos exteriores representan a las clases involucradas. Hay que hacer notar que se ha hecho una simplificación: había más de dos clases involucradas, pero esta modificación no

altera el interés del ejemplo, que por el contrario queda mucho más simple y legible. Los rectángulos internos representan operaciones (es decir, métodos de clases), y las flechas representan la estructura de llamadas.

Las operaciones que aparecen con fondo gris, junto con las flechas gruesas, representan la secuencia de llamadas que era necesario modificar. Había que derivar una nueva subclase de `CDocument` (en términos de programación C++) e implementar una nueva versión para la operación `OnSaveDocument()`. Además de eso, el comportamiento original de `DoFileSave()` y `DoSave()` no era apropiado.

La solución apropiada habría sido redefinir también estas dos funciones en la clase derivada, `CMyDocument`. Pero existía un problema para ello: el diseñador de la biblioteca de clases no había considerado a estas funciones como virtuales, y en consecuencia no estaban declaradas con la palabra reservada `virtual`. Este es un problema muy común al usar marcos de referencia desarrollados en C++; sus diseñadores no pueden anticipar todos los posibles usos o modificaciones de la estructura original, y a la vez los usuarios se ven obligados a adaptarse a un diseño ya existente, incluso cuando ese diseño no es el más apropiado para sus propósitos. No tienen la posibilidad de rehacer el código para tener en cuenta las necesidades actuales.

Ante esta situación, puede pensarse que el programador debería modificar de todas formas la biblioteca de clases y convertir esas funciones en virtuales, con el fin de salvaguardar la calidad. Pero esta opción no era aceptable. Aunque la biblioteca MFC se suministra con código fuente, el crear una recompilación propia de esta biblioteca –compartida por muchas aplicaciones- podía traer multitud de problemas de distribución e instalación, o incluso romper el funcionamiento de muchas aplicaciones de terceros. Téngase en cuenta que el convertir un método en virtual altera la tabla de funciones virtuales, rompiendo la compatibilidad binaria entre módulos; dicho de otro modo, esa modificación habría exigido la recompilación de todas las aplicaciones basadas en MFC, cosa de todo punto imposible en la práctica.

Así que el programador decidió adoptar un enfoque alternativo. Era necesario evitar los efectos indeseados de `DoFileSave()`, así que se modificó `OnFileSave()` (que por fortuna sí estaba declarada como virtual) para que invocase directamente a `OnSaveDocument()`. Para sustituir la parte de funcionalidad de estos métodos que sí era necesario conservar, `OnSaveDocument()` tenía que pasar a invocar a `DoPromptFileName()` como hacía `DoSave()`), introduciendo una nueva dependencia. Esta estructura modificada se representa en la Ilustración 24. Evidentemente, el diseño descrito conlleva riesgos. El programador se ha visto obligado a introducir dependencias que no existían en el diseño original de la biblioteca, y a alterar su comportamiento habitual de una forma que sus diseñadores no habían previsto. La evolución y el mantenimiento del sistema se ven, por tanto, comprometidos. Incluso si los supuestos del programador se recogen cuidadosamente en algún documento de diseño, esta información no se podría verificar de forma automática en el futuro.

Este es un caso en el que la aplicación de Itacio bajo la interpretación de los contratos de reutilización puede ser útil. En primer lugar, se puede documentar la estructura original del sistema de grabación de documentos de MFC mediante un contrato de reutilización que describa la interacción entre los participantes; véase en la Ilustración 25 el esquema del contrato y en la Ilustración 26 su expresión en Itacio.

| | |
|---|---|
| **PARTICIPANT**: cDocument<br>**OPERATIONS:**<br>::doFileSave()<br>  cDocument::doSave()<br>::doPromptFileName()<br>::doSave()<br>  cDocument::doPromptFileName()<br>  cDocument::onSaveDocument()<br>::onFileSave()<br>  cDocument::doFileSave()<br>::onFileSaveAs()<br>  cDocument::doSave()<br>::onSaveDocument()<br>::saveModified()<br>  cDocument::doFileSave() | **PARTICIPANT**: msgRouter<br>**OPERATIONS**:<br>::userPushesSave()<br>  cDocument::onFileSave()<br>::userPushesSaveAs()<br>  cDocument::onFileSaveAs()<br><br>**ACQUAINTANCES**:<br>cDocument::me->cDocument<br>msgRouter::targetDoc->cDocument |

*Ilustración 25. Esquema original de grabación de documentos de MFC descrito como un contrato de reutilización.*

Con este punto de partida, las adaptaciones que se realizan a este esquema se pueden representar también como operadores de contratos. En este ejemplo, los operadores canónicos se aplican tal cual, aunque a efectos prácticos sería frecuente empaquetar en forma de operadores combinados predefinidos las secuencias de modificación utilizadas frecuentemente.

```
Type=saveMFC
Sources=res
BEGIN_RESTRICTIONS

isContract($res$).

participant($res$, msgRouter).
participant($res$, cDocument).

acqRelationship($res$, msgRouter, targetDoc, cDocument).
acqRelationship($res$, cDocument, me, cDocument).

operation($res$, cDocument, onFileSave).
operation($res$, cDocument, onFileSaveAs).
operation($res$, cDocument, saveModified).
operation($res$, cDocument, doFileSave).
operation($res$, cDocument, doSave).
operation($res$, cDocument, doPromptFileName).
operation($res$, cDocument, onSaveDocument).

operation($res$, msgRouter, userPushesSave).
operation($res$, msgRouter, userPushesSaveAs).

operationInvocation($res$, msgRouter, userPushesSave, targetDoc,
   onFileSave).
operationInvocation($res$, msgRouter, userPushesSaveAs,
   targetDoc, onFileSaveAs).

operationInvocation($res$, cDocument, saveModified, me,
   doFileSave).
operationInvocation($res$, cDocument, onFileSave, me,
   doFileSave).
operationInvocation($res$, cDocument, doFileSave, me, doSave).
operationInvocation($res$, cDocument, onFileSaveAs, me, doSave).
operationInvocation($res$, cDocument, doSave, me,
   doPromptFileName).
operationInvocation($res$, cDocument, doSave, me,
   onSaveDocument).

END_RESTRICTIONS
```

*Ilustración 26. El contrato de la Ilustración 25 codificado como un componente de Itacio.*

En el trabajo de Lucas, se propone que la combinación de varios operadores debe evaluarse bajo ciertas reglas de compatibilidad para prevenir la aparición de contratos mal formados; pero el modelo de inferencia de Itacio ofrece una ventaja significativa. Verifica el sistema utilizando toda la información que existe sobre el mismo [CLC01a] y no son necesarias esas "recetas" específicas. En la Ilustración 27 puede verse un grafo, generado en PostScript por el prototipo Itacio-XDB, en el que se recoge la cadena de modificaciones que conduce del esquema de la Ilustración 23 al de la Ilustración 24.

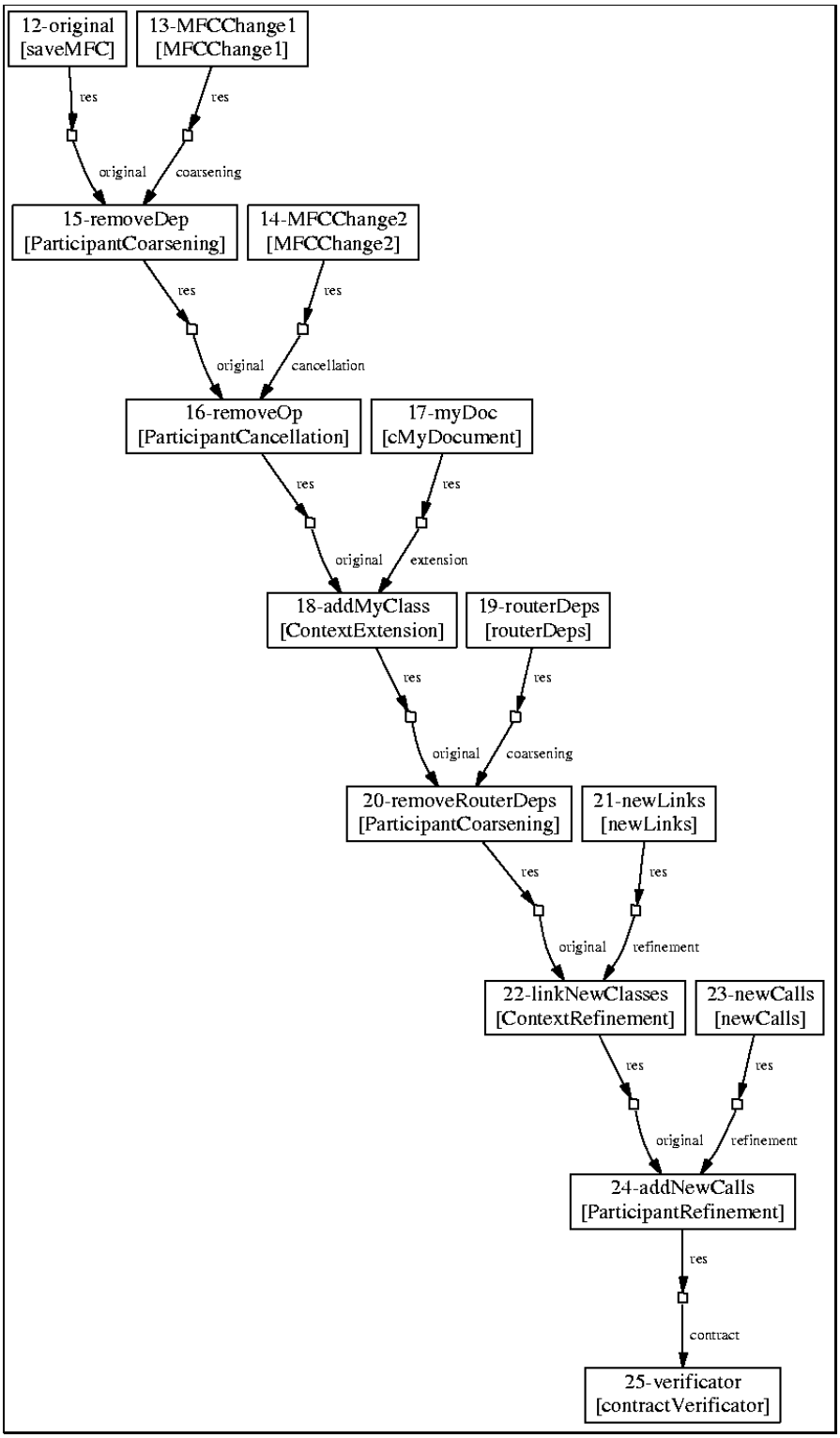



*Ilustración 27. El sistema de grabación de documentos de MFC y sus modificaciones, modelados en Itacio-XDB bajo la interpretación de contratos de reutilización (versión grafo PostScript)*

El componente de arriba a la izquierda de la Ilustración 27 (el número 12) representa el contrato que describe el comportamiento original de MFC, y los demás rectángulos representan la cadena de modificaciones hasta llegar a la estructura deseada (abajo a la derecha, en la fuente del componente 24). Como ya se ha dicho, el número de cajas en la imagen podría ser mucho menor si se utilizasen operadores combinados (lo que no se ha hecho por generalidad, por representar fielmente el modelo de Lucas y porque no tiene influencia en nuestros objetivos de investigación). Finalmente, el componente número 25 tiene como requisito en su sumidero *contract* que dicho contrato esté bien formado.

Esa estructura representa, de hecho, una base de conocimientos que puede utilizarse para razonar sobre el sistema. Por ejemplo, una nueva versión de MFC podría incorporar cambios tales como eliminar la operación `DoPromptFileName()` (véase Ilustración 24). Para evaluar el impacto de este cambio en el sistema (nuestro editor gráfico) que hemos construido sobre MFC, no es necesario realizar una revisión manual completa contrastando los cambios en MFC con la documentación de diseño. Bastaría con aplicar un operador al contrato base de MFC para representar el nuevo comportamiento (o bien modificar directamente el contrato base, si el usuario no está interesado en representar explícitamente la evolución de MFC sino trabajar sólo con la versión más reciente). Aunque la modificación se realiza al principio de la cadena, el sistema señala el problema (con un cuadrado rojo de mayor tamaño) donde se manifiesta: en este caso, aunque cada una de las modificaciones por sí sola es legal y aplicable, el resultado de la modificación es un contrato que no está bien formado (Ilustración 28).

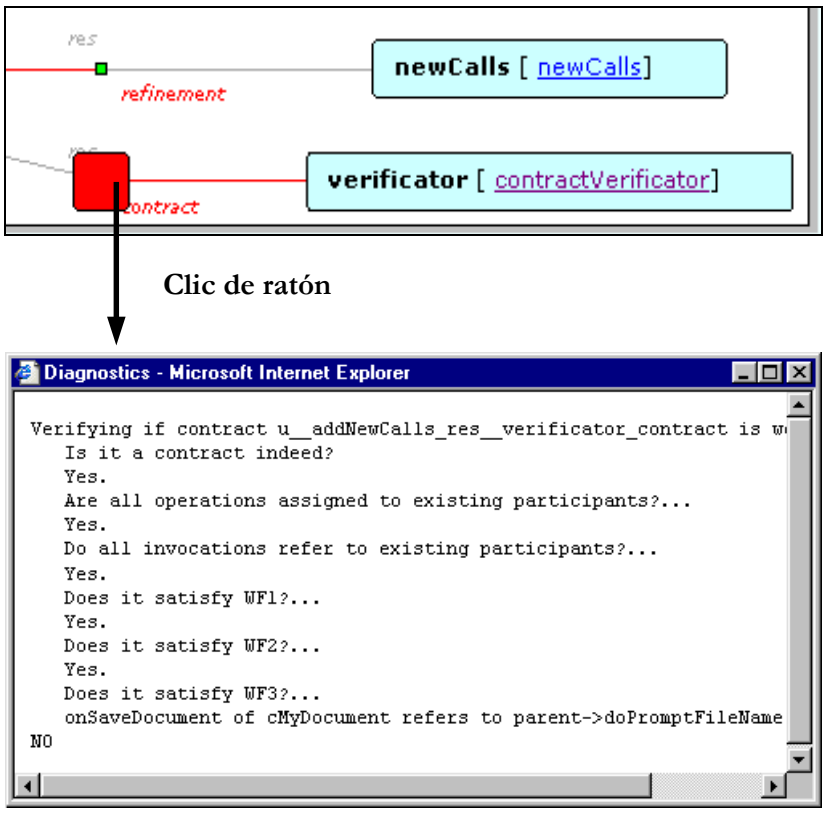



*Ilustración 28. Detectando una inconsistencia en la evolución de un contrato (captura de pantalla del prototipo Itacio-XJ).*

Itacio es capaz de explicar el problema. En este caso, el cambio descrito en la implementación de MFC haría fallar nuestro código, porque se han realizado suposiciones cuando se construyó. Estas suposiciones pueden dejar de ser ciertas cuando MFC se ve modificada. El documentar estas suposiciones con contratos de reutilización expresados en Itacio constituye una buena forma de detectar este tipo de problemas de manera general.

La eliminación de una operación de una clase base parece una modificación radical, pero no sería lo único que pudiese detectarse. De hecho, el modelo Itacio es extremadamente flexible, y cualquier restricción susceptible de ser expresada mediante cláusulas Horn puede ser tenida en cuenta durante el proceso de verificación.

Por ejemplo, supóngase que la operación `DoPromptFileName()` (véase de nuevo la Ilustración 23) no se elimina en la siguiente versión de MFC, pero `DoSave()` deja de invocarla. Puesto que nuestra versión modificada de `OnSaveDocument()` está intentando reproducir el comportamiento de `DoSave()` (y de hecho esta es la razón por la cual `OnSaveDocument()` invoca a `DoPromptFileName()`), el hecho de que `DoSave()` deja de realizar esta invocación es una información muy importante. Este cambio no rompería la estructura básica de los contratos, pero invalidaría una suposición fundamental.

El modelo original de contratos de Lucas no está diseñado para expresar una restricción como "Este contrato requiere que en el contrato base la operación A invoque a la operación B"; no obstante, este requisito se puede expresar en Itacio con una sola sentencia declarativa, sin que se requiera una adaptación específica del sistema. Si la invocación de la operación se elimina del contrato base de MFC, el sistema señalará dónde se viola el requisito exactamente igual que en el caso de la eliminación de operaciones, y explicando también las razones específicas del problema.

**Conclusiones**

El modelo de contratos de reutilización es una estructura conceptual válida sobre la que se pueden realizar razonamientos. Estos razonamientos pueden llevarse a cabo de manera estática, sin necesidad de probar o ejecutar (ni siquiera construir) el programa que se está inspeccionando.

El modelo Itacio se suponía lo bastante flexible para encajar con diferentes nociones de componente y expresar muy diversos tipos de restricciones. La aplicación del sistema de verificación de Itacio a los contratos de reutilización ha sido una demostración de este hecho, puesto que Itacio no se había diseñado para ser aplicado en este campo concreto.

La combinación de contratos de reutilización y el modelo Itacio puede resultar valiosa (siempre y cuando se desarrollen herramientas apropiadas, lo que parece perfectamente posible a juzgar por la simplicidad de los prototipos utilizados en los experimentos) para verificar la evolución de un sistema. Permite modelar la estructura de colaboraciones del mismo, recoger los supuestos del diseñador y verificar automáticamente que estos supuestos y requisitos se cumplen.

### 5.2.3. Diagnóstico remoto de equipos

En este apartado se presenta un caso de uso de Itacio basado en una interpretación del concepto de componente software que se acerca a la tradicional, pero aun así no es el caso

típico. Se trata de considerar componentes a elementos software ya existentes en el ordenador (tales como aplicaciones completas) y utilizar el modelo de verificación para comprobar que un equipo concreto se encuentra en un estado adecuado (en el que los diversos elementos implicados pueden interoperar con arreglo a sus respectivos requisitos).

**El diagnóstico de equipos**

El usuario medio dispone actualmente de sistemas informáticos muy potentes, pero el software involucrado es muy complejo. Son frecuentes (sobre todo en plataformas Windows, muy extendidas en el ámbito de la informática personal) los problemas de instalación, configuración, interacción entre diferentes versiones de componentes software, etc. El diagnóstico de estos problemas puede ser costoso, y es necesario un conocimiento deductivo especializado, además del escrutinio de ciertas características difícilmente manejables por el usuario (fechas de archivos, versiones, claves del registro del sistema...)

Aun en el caso de equipos que funcionan correctamente, en ocasiones cobra importancia el mero hecho de que la configuración del equipo responda a cierto modelo predeterminado. El personal de tecnologías de la información conoce bien el esfuerzo que requiere mantener una sincronización entre versiones y productos que se respete en todos los ordenadores de una empresa, respondiendo a la política de medios informáticos establecida. Con el fin de evitar trabajo duplicado y desplazamientos de este personal, se han desarrollado diversas herramientas de "acción a distancia" (como pcAnywhere, de la casa Symantec [Syma], o VNC, desarrollado por AT&T [VNC]) y de duplicación de discos duros (como Norton Ghost, también de Symantec), cuyo uso puede representar un considerable ahorro en tiempo y esfuerzo.

Teniendo en cuenta este panorama, parece útil disponer de alguna herramienta capaz de realizar a distancia verificaciones sobre el estado de un ordenador personal, verificaciones que cuando son llevadas a cabo por un técnico son rutinarias y repetitivas. El objetivo aquí es poder describir de alguna forma la configuración que se desea para un equipo, y que un sistema de verificación compruebe en qué grado se ajusta un equipo real a esa descripción. Esto podría resultar de interés, entre otros, para:

- Empresas de desarrollo de software interesadas en automatizar la verificación de la instalación de sus productos y el diagnóstico de problemas de configuración de software
- Empresas de desarrollo de software interesadas en desarrollar soluciones estándar para el diagnóstico de equipos
- Empresas de mantenimiento de equipos informáticos, software, atención al usuario... interesadas en automatizar tareas de diagnóstico y ayuda al usuario
- Empresas de cualquier tipo, con departamento de Informática y cierto volumen de usuarios de informática (industria, banca, redes de franquicias, etc.), interesadas en automatizar el mantenimiento y verificación de sus equipos (especialmente si estos se encuentran geográficamente dispersos)

**Aplicación de Itacio al diagnóstico remoto**

Mediante un modelo de componentes basado en conocimiento como es Itacio, es posible describir los diferentes componentes software asociándoles sus "reglas de buen

funcionamiento". Esto permitirá generar una base de conocimientos sobre el sistema, que puede entonces ser interrogada (mediante el software de inferencia de ECLiPSe) en busca de incoherencias o fricciones entre los componentes (es decir, de incumplimientos de sus requisitos de funcionamiento).

En este caso, existe un problema adicional a resolver. El modelo de verificación de Itacio, por sí solo, podría utilizarse sólo para verificar la coherencia de un modelo de componentes en el plano teórico, es decir, como diseño. Pero en este caso se desea que el sistema verifique *realmente* la configuración de un PC determinado. Es decir, además de comprobar que efectivamente los requisitos de los componentes involucrados se satisfacen en un plano teórico, se desea comprobar que en la práctica se están cumpliendo también dichos requisitos.

En otros casos de aplicación estudiados en esta tesis, el uso de Itacio se limita a verificar el modelo del sistema, tomando las especificaciones de sus componentes; no se conecta este modelo con elementos "reales". Este problema se ha asumido en el planteamiento de Itacio cuando se hace referencia a código fuente. Hay un doble motivo para esto. El primero es que el análisis del mismo para comprobar que los hechos que se dan por ciertos en la base de conocimientos son efectivamente ciertos en el propio programa suele resultar excesivamente complejo para las restricciones de sencillez y viabilidad que afectan a esta tesis. El segundo, que el objetivo de la tesis es hacer uso del conocimiento disponible, y aunque sería positivo verificar en la realidad ese conocimiento para descartar datos erróneos (ejemplo: un hecho que afirma que cierto programa termina, cuando en realidad el programa puede no terminar nunca debido a un error), el objetivo se cumple aun sin esa posibilidad.

No obstante, en este caso no hay por qué asumir esa separación entre modelo y realidad, ya que a diferencia de lo que ocurría con el análisis del código fuente, el análisis –mediante software- de un ordenador para verificar la verdad o falsedad de hechos tales como la presencia o ausencia de un fichero sí que es plenamente viable. Tómese como ejemplo una biblioteca (DLL) que forme parte del sistema que se modela, y cuya presencia se requiere como requisito. Aunque Prolog y sus extensiones CLP no puedan verificar este hecho de manera nativa, pueden invocar a bibliotecas que sí tengan esa capacidad.

Por tanto, es técnicamente viable asociar esas reglas o hechos con programas que comprueben realmente lo que se necesita saber del equipo, un *software de evaluación de síntomas*. Si esos programas de diagnóstico son accesibles a través de la red (Internet o intranet), el resultado es que el conocimiento deductivo para detectar averías puede encontrarse en un servidor centralizado, mientras que el equipo que se examina sólo necesita descargar un pequeño programa evaluador de síntomas que actúa como "operario" del sistema de diagnóstico. De este modo es posible automatizar multitud de diagnósticos, y el usuario puede comprobar su equipo sin necesidad de que personal especializado se desplace constantemente y realice las mismas verificaciones una y otra vez. La posibilidad de realizar estas verificaciones de forma automatizada ha sido comprobada repetidamente por parte del autor; el proyecto AWDiagnostico, desarrollado en la empresa Seresco, S.A., verifica la instalación de un sistema de edición de documentos cliente/servidor y diagnostica de manera dinámica los errores de ejecución ofreciendo explicaciones que dependen de la versión concreta del software de ofimática instalado. En [Ce01b] pueden verse detalles sobre el desarrollo de utilidades de diagnóstico similares.

Este es el enfoque adoptado aquí, y que permite que el proceso de verificación no sólo compruebe la bondad de un modelo, sino el hecho mismo de que ese modelo se está cumpliendo y tiene fiel reflejo en la realidad.

**Estructura del sistema**

Dado el modelo habitual de operación de Itacio, la única novedad es que algunos predicados de las expresiones restrictivas no son ciertos ni falsos sin más, sino que recaban datos sobre un ordenador real. Para ello, los predicados invocan funciones externas que se encuentran en la biblioteca TCPProxy.DLL, desarrollada para este propósito.

Al principio de la base de conocimientos que deba invocar a TCPProxy.DLL, figuran los predicados siguientes:

```
:- load('tcpproxy.dll').
:- external(makeRequest/3, p_makeRequest).
```

El primero carga en memoria la biblioteca en cuestión, y el segundo declara como externo el predicado `makeRequest/3`, que se corresponde con la función `p_makeRequest` de TCPProxy.DLL. De hecho, la única función exportada por esta biblioteca es `int p_makeRequest()`.

TCPProxy.DLL ha sido desarrollada en lenguaje C. Aunque según la declaración anterior el predicado `makeRequest` tiene aridad 3, es decir, recibe tres parámetros, la función `p_makeRequest` no recibe ninguno. La explicación a esto es que las funciones invocadas desde ECLiPSe no reciben los parámetros de la forma habitual en C, sino que acceden a los mismos mediante ciertas funciones específicas suministradas por ECLiPSe. En el cuerpo de `p_makeRequest()`, pues, se realizan llamadas al runtime de ECLiPSe para poder obtener los valores de los parámetros (y asignar valores a los parámetros de salida).

Resuelta la comunicación entre el código ECLiPSe y el código C, queda la verificación de los hechos en cuestión. Puesto que la instalación de todo el sistema de deducción de Itacio-XDB en el equipo a verificar resultaría muy costosa, además de perturbar notablemente la configuración del mismo, se ha desarrollado un sistema gracias al cual la verificación se realiza a través de TCP/IP. El equipo objeto de la verificación sólo necesita tener en ejecución un pequeño programa servidor, denominado TCPStub.EXE; la función `p_makeRequest()` de TCPProxy.DLL se comunica a través de la red con este servidor, y le remite las *preguntas* concretas.

El protocolo de comunicación entre TCPProxy.DLL y TCPStub.EXE, en aras de la simplicidad y flexibilidad, consiste en el envío de una sola cadena a TCPStub, cuyos elementos están separados por el símbolo "|", y el retorno de una cadena de este a TCPProxy con los resultados. Como ilustración del tipo de información que TCPProxy / TCPStub son capaces de dar, se ofrece en la Tabla 6 un listado de los códigos de petición posibles. (Para una mejor comprensión de la terminología de esta tabla, se sugiere recurrir a [Pe98]).

| Orden | Parámetros | Efecto |
|---|---|---|
| `END` | - | Termina la ejecución de TCPProxy. |
| `SEARCHFILE` | Nombre de fichero | Busca en el disco duro el fichero solicitado, y devuelve los directorios en los que apareció, con ruta completa y separados por punto y coma. |
| `LOADMODULE` | Nombre de fichero | Siendo el fichero un módulo ejecutable, lo busca en una serie de directorios establecidos en la configuración del sistema, intenta cargarlo en memoria, y devuelve información sobre si lo pudo cargar en memoria o no, el motivo del error (si no pudo), el nombre del fichero con ruta completa, la versión del fichero y el idioma. |
| `OS` | - | Devuelve el nombre del sistema operativo (95, 98, NT-2000) y los números de versión del mismo. |
| `REGVALUE` | Clave raíz<br>Clave<br>Valor buscado | Devuelve el valor del *registry* que se le solicita, o un código de error si no existe. Se supone que el valor es de tipo cadena. |
| `WINDIR` | - | Devuelve cuál es el directorio Windows del sistema. |
| `SYSDIR` | - | Devuelve cuál es el directorio System del sistema. |
| `INIVALUE` | Fichero INI<br>Sección<br>Clave | Devuelve el valor del fichero INI que se le pide. |
| `LASTLINEOFFILE` | Nombre de fichero | Devuelve la última línea de un fichero de texto dado. |
| `FILEDATE` | Nombre de fichero | Devuelve la fecha de última modificación del fichero cuyo nombre recibe como parámetro. |
| `ENVVAR` | Nombre de variable | Devuelve el valor de una variable de entorno. |

*Tabla 6. Peticiones aceptadas por TCPProxy.EXE*

Puede verse un ejemplo de esta infraestructura en la Ilustración 29. En este caso, existiría un modelo de un componente, `MFC Runtime Library`; este componente representa a la biblioteca para tiempo de ejecución de MFC [Po99]. Dicho componente presenta en este caso una fuente, la cual lleva asociadas algunas garantías; entre ellas, la que aparece en la figura. Esta garantía establece un caso en el cual la biblioteca de MFC.DLL ofrece soporte para OLE [Br93]. En la evolución de MFC, las clases que daban soporte a OLE aparecieron en la versión 3.0 de la misma; por tanto, esta garantía intenta verificar este hecho. Intenta cargar la biblioteca en memoria (en la memoria del equipo diagnosticado), obteniendo la cadena que representa a la versión; acto seguido, extrae de esa cadena el número principal de versión (conocido como *major versión number*), y por último exige que ese número sea 3 ó superior. Si se dan las condiciones, el predicado `ole_support` pasará a ser cierto para las características de esa biblioteca.

Lógicamente, la invocación al predicado `td_load_module` no podría verificar realmente la situación del equipo diagnosticado si no existiese una interacción con el mismo. Esto se consigue porque `td_load_module` invoca al predicado externo `makeRequest` que ya se ha mencionado anteriormente; dicho predicado externo se encuentra implementado en TCPProxy.DLL, que se comunica vía TCP/IP con TCPStub.EXE, si este se encuentra en ejecución en el equipo diagnosticado. TCPStub realiza la tarea encomendada (cargar MFC.DLL, extraer información sobre su versión, etc.) y se la devuelve a TCPProxy.DLL, que a su vez remite los resultados al predicado que le había invocado, `td_load_module`. Este predicado pasa a ser cierto y las variables se han unificado con los valores correspondientes. Gracias a este proceso, las expresiones restrictivas de componentes como el `MFC Runtime Library` de ejemplo no sólo modelan su comportamiento en un plano teórico, sino que su evaluación reproduce la situación real; las *interpretaciones* de estos predicados se ajustan al estado del ordenador que se verifica.

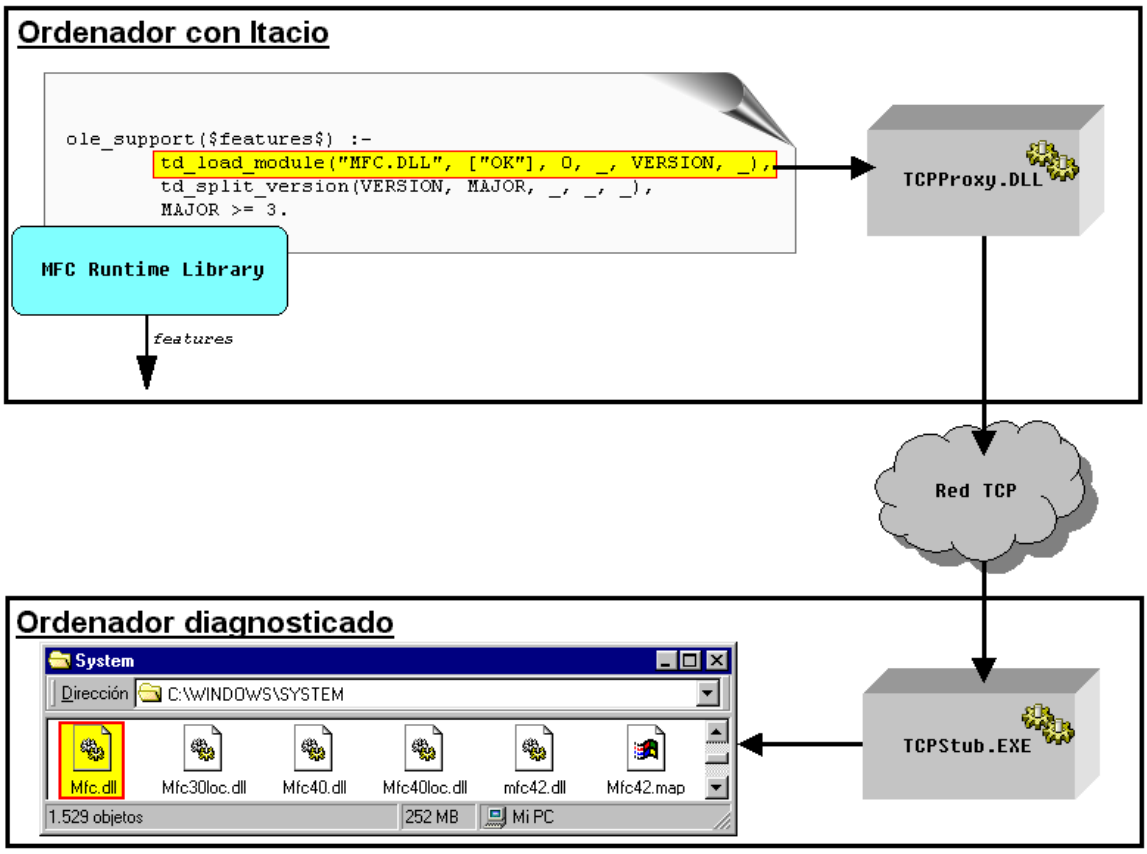



*Ilustración 29. Itacio aplicado al diagnóstico remoto de equipos.*

### Un problema de ejemplo

Una vez presentada la estructura básica que permite a Itacio evaluar realmente el estado de un equipo, podemos pasar a un ejemplo de aplicación, como se ha hecho en otros experimentos recogidos en esta tesis.

Un caso típico de verificación del estado de un equipo es el comprobar el software instalado de cara a la realización de un trabajo determinado; por ejemplo, de cara a realizar estudios de una asignatura de programación en Informática. Normalmente, se establecen ciertas herramientas básicas y ciertas actividades que el alumno deberá poder realizar con el material disponible.

Una posible **instanciación** del modelo Itacio es, pues, considerar que la estación de trabajo del alumno es el sistema que se modela, y que esta estación debe incluir diversos sistemas y subsistemas. Haciendo nuevamente uso de la definición flexible de componente software que aquí se maneja, y puesto que nuestro interés es el modelado *estructural* del software presente en esta estación de trabajo, dicha estación constaría de diversos subsistemas, que a su vez incluirían aplicaciones para las tareas concretas. En este caso, las entradas y salidas de estos componentes –subsistemas, aplicaciones- tienen como misión transmitir información sobre el estado de los mismos; no tratan de reproducir su comportamiento, sino su situación y configuración. Por ejemplo, si uno de los componentes es un procesador de textos, seguramente en otro ámbito podría tener como entrada la pulsación de una tecla, o como salida una versión PostScript del documento; pero bajo el enfoque que ahora nos ocupa este componente ofrecerá en sus salidas información sobre el fabricante, la versión, directorio de instalación, y otros aspectos estructurales similares. En este caso no tiene sentido la semántica de invocación de métodos que frecuentemente se aplica a las entradas y salidas de los componentes.

Yendo más al detalle, se puede idear un supuesto según el cual la estación de trabajo de un alumno debería presentar una estructura como la siguiente:

- Soporte para tareas de desarrollo (programación). Esto incluye los compiladores de los diversos lenguajes a emplear:
    - Un compilador de C++. El compilador aceptado para esta asignatura sería Microsoft Visual C++, versión 6.0 ó superior.
    - Un compilador de Pascal. El compilador aceptado sería Borland Delphi, versión 5.0 ó superior.
    - Un compilador de Java. Esto exige la presencia del Java Development Kit (JDK), versión 1.3 ó superior.
- Soporte para tareas de escritura de documentos. Esto incluye:
    - Un procesador de textos. El procesador admitido es Microsoft Word, no importa qué versión.
    - Una herramienta para crear diagramas de análisis orientado a objetos. En principio, tendría que ser Rational Rose, no importa qué versión.
- Soporte para leer documentos, lo que incluye:
    - Un visor de documentos Word. En principio, aunque existen otros visores sin capacidad de edición, sólo se ha modelado Microsoft Word, y se exige que sea la versión 2000 o superior.
    - Un visor de documentos web. Sólo se ha modelado Internet Explorer, que debe ser la versión 4.0 ó superior y haber instalado además soporte para gráficos vectoriales en formato VML.

- Un visor de documentos PDF, no importa qué versión; según el modelo, puede ser el Adobe Acrobat Reader o el Aladdin Ghostscript.
- Un visor de documentos PostScript. Sólo se ha modelado el Ghostscript.

Cabe hacer algunos comentarios sobre esta relación de componentes. En algunos casos, el modelo es muy limitado; téngase en cuenta por ejemplo que sólo se acepta como compilador de Pascal el Delphi de Borland. Esto no debe mover a confusión; es perfectamente posible modelar otras alternativas mediante las expresiones restrictivas de Itacio. Bastaría con recopilar información sobre cómo detectar que se hallan instalados otros productos, y esta es una tarea rutinaria que no pone en cuestión la capacidad del prototipo. Lo contrario daría un ejemplo más complejo, que no aportaría beneficios notables a este texto. Otro comentario cabe hacer sobre el procesador de textos; se exige una versión avanzada del lector, ya que debe ser capaz de presentar documentos de variada procedencia (es decir, de cualquier versión) pero se acepta que el procesador de escritura sea de versiones más viejas, precisamente porque los demás alumnos o profesores podrán leerlo en virtud del lector avanzado que tendrán que tener instalado.

Con esta información, y mediante los recursos que brindan TCPProxy y TCPStub, basta con averiguar cómo obtener información sobre los productos que se encuentran instalados (cosa que la mayoría de las veces se resuelve con simples consultas al *registry* del sistema) y redactar las expresiones restrictivas en consonancia con esto. Así se ha hecho, y la aplicación de este prototipo a un PC elegido al azar (que correspondía a un profesor que no impartía esta asignatura) revela los incumplimientos del modelo que aparecen en la Ilustración 30. En efecto, una comprobación manual del equipo reveló que el diagnóstico era correcto. Este equipo no contaba con Delphi ni Visual C++. En el caso de Java, el sistema diagnosticó que no estaba presente ninguna versión del JDK, como así era.

Por lo que se refiere a Rational Rose, tampoco este producto estaba instalado; y lo mismo ocurría con un visor de PostScript (el usuario de este equipo podía leer PostScript, pero transformándolos previamente a PDF con una utilidad que no estaba prevista en el modelo del sistema). Los demás componentes eran correctos; al contar ese PC con Microsoft Word 2000, se satisfacían los requisitos de lectura y escritura de documentos. Alterando manualmente la configuración, se pudo comprobar que si el procesador de textos instalado hubiese sido, por ejemplo, Microsoft Word 97, el sistema daría por buenas las características de Word de cara a la escritura de documentos, pero señalaría una violación de restricciones en su conexión con el subsistema de lectura de documentos.

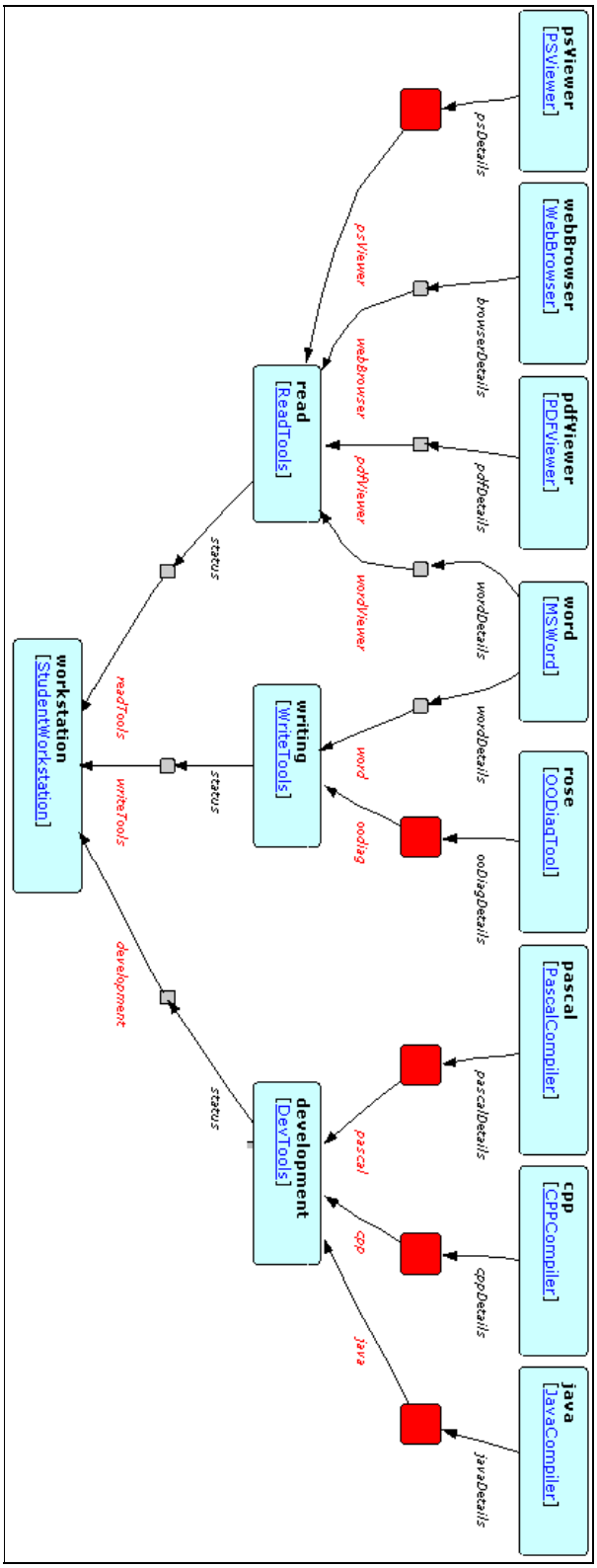



*Ilustración 30. Itacio-XDB verificando la idoneidad de un PC concreto para el trabajo en cierta asignatura de Informática.*

**Conclusiones**

Nuevamente, se ha sometido al modelo Itacio a un campo de aplicación inicialmente imprevisto, que consiste en considerar los componentes de un sistema desde un punto de vista estructural y estático: las versiones, compatibilidades, presencia o ausencia de los mismos, etc. Yendo más allá del modelado, el uso de bibliotecas externas a ECLiPSe permite ligar el proceso deductivo a procedimientos de programación imperativa, haciendo que los predicados tengan una interpretación real, ya que su valor de verdad o falsedad puede averiguarse "en el mundo real". Esto permite modelar sistemas reales y verificar el cumplimiento del modelo en la realidad.

Estas sencillas extensiones permiten llevar el prototipo Itacio al estatus de aplicación práctica, más allá de los experimentos de laboratorio (y de hecho el diagnóstico remoto de equipos mediante Itacio se ha incluido en la Oferta Tecnológica de la Universidad de Oviedo en su edición de 2001 [Ce01]). Se ha podido verificar una vez más la flexibilidad del modelo, su utilidad en la práctica, y la facilidad de desarrollo de las aplicaciones basadas en Itacio, ya que el diagnóstico remoto se ha resuelto con un limitado esfuerzo de programación y haciendo uso de una herramienta –el prototipo Itacio-XDB- ciertamente sencilla y sin pretensiones –como prototipo que es, sin capacidades de edición gráfica o similares-.

## 5.2.4. Sistema de procesamiento de sonido en tiempo real basado en componentes: WaveX

Con el fin de demostrar la flexibilidad y aplicabilidad de Itacio en ámbitos diversos, en esta tesis se presentan casos en los que el concepto de componente software se utiliza de forma muy abierta y se aplica a entes que no son considerados habitualmente como tales componentes. Evidentemente, también conviene experimentar con la aplicación de Itacio a un esquema de componentes tradicional, como el presentado en la Def. I y siguientes.

Así se ha hecho en el proyecto WaveX, un sistema basado en componentes que podríamos denominar "tradicional" por lo que se refiere al concepto de componente que se maneja, pero con restricciones específicas. Estas restricciones sirven perfectamente de ejemplo para recoger la enorme variedad de requisitos que pueden plantearse en el mundo real, y que los artefactos utilizados frecuentemente con componentes software típicos (como el IDL o similares) no pueden recoger ni verificar.

### El sistema de procesamiento de sonido WaveX

WaveX es un sistema de procesamiento de sonido en tiempo real desarrollado en C++, haciendo uso del compilador Microsoft Visual C++ 6.0. Las *Microsoft Foundation Classes* (MFC) mencionadas en el capítulo 5.2.2 se utilizan también, sobre todo para la interfaz gráfica con el usuario.

El objetivo de WaveX es ofrecer un sistema de procesamiento de sonido que aproveche la potencia de proceso de los ordenadores personales modernos. Los dispositivos de procesamiento de sonido profesionales son caros, mientras que los ordenadores personales actuales han alcanzado una potencia de proceso que posibilita el proceso de sonido en tiempo real mediante software. Además, los dispositivos de grabación y reproducción para ordenadores personales (dicho de otro modo, las tarjetas de sonido) están muy extendidas y resultan asequibles; de hecho, existen placas base que incorporan estas capacidades en su propio diseño, sin necesidad de tarjetas adicionales. En el mercado existen ya dispositivos de procesamiento de sonido que se basan en ordenadores personales, pero es habitual que requieran del uso de hardware adicional [Kyma, Waves]. También existen muchos otros productos para edición de audio en ordenadores personales, pero no es frecuente que estén orientados al procesamiento en tiempo real [Gold].

Por lo tanto, el uso de ordenadores domésticos de propósito general en lugar de dispositivos de procesamiento de sonido específicos puede ser una alternativa barata y adecuada para muchos usuarios. Lo que se hace en estos programas es digitalizar algunos milisegundos de sonido (mediante un conversor analógico-digital como el que incorporan las mencionadas tarjetas de sonido) y almacenar ese sonido en un buffer, en forma de valores discretos que representan la amplitud de la señal a intervalos regulares de tiempo; el procesamiento de sonido puede realizarse entonces en forma numérica sobre estos valores, y el hardware conversor digital-analógico genera la onda analógica resultante que podemos oír [Si97]. La considerable capacidad de proceso de los ordenadores personales modernos permite realizar estas transformaciones de manera lo suficientemente rápida como para proporcionar una fuente constante de sonido, en tiempo real; basta con repetir las mismas operaciones con el siguiente buffer mientras el anterior está siendo reproducido por el

hardware de sonido, sin interrupciones. La condición que debe cumplirse para ello es que el tiempo necesario para procesar un buffer debe ser inferior a la duración del mismo.

En general, el procesamiento de sonido se realiza combinando diferentes etapas, que en la práctica se corresponden con dispositivos (tales como pedales de distorsión, mezcladores y similares) cuyas entradas y salidas se conectan mediante cables. Puesto que WaveX pretende sustituir esta estructura mediante software, parecía lógico pensar en un desarrollo basado en componentes. Por tanto, se definió un marco general para la interconexión, y se pasó a desarrollar un módulo o componente (en el sentido tradicional del término) para cada efecto deseado. El usuario de WaveX puede describir la llamada **topología** del procesador de sonido en un fichero de texto (cuya estructura se detalla más adelante) y cuando este fichero se carga en WaveX el sistema crea instancias de todos los componentes necesarios, conectándolos de manera adecuada. Por supuesto, la flexibilidad intrínseca del software es una gran ventaja, ya que se pueden desarrollar y utilizar nuevos módulos para efectos específicos a un coste relativamente bajo.

Se decidió construir WaveX como una aplicación para el sistema operativo Microsoft Windows, en la que los módulos serían Bibliotecas de Enlace Dinámico (*Dynamic Link Libraries*, DLLs [Pe98]) con una interfaz concreta; se definió un esquema de interfaz propio para dichas DLLs. No se recurrió a *middleware* (como COM o CORBA) por varias razones. Primero, la eficiencia y la simplicidad de desarrollo eran factores importantes. Por tanto, el tiempo extra de las llamadas a través de middleware no era deseable, ni lo era tampoco la mayor complejidad de desarrollo que normalmente implican las plataformas mencionadas. Pero, sobre todo, el middleware no habría proporcionado ningún beneficio notable en este caso. No había planes de comunicación entre procesos ni entre máquinas, ni de instalación distribuida; tampoco se deseaba soporte para múltiples lenguajes de programación.

La versión inicial de WaveX incluye varios componentes. Puesto que el objetivo de esta tesis es el modelo Itacio y no este sistema específico, se describirán aquí sólo algunos de los componentes, por razones de simplicidad; sólo se pretende proporcionar una comprensión básica de lo que el sistema puede hacer.

**DV_WaveInDevice** captura el sonido que la tarjeta de sonido está digitalizando y lo suministra como la única salida del componente.

**DV_WaveOutDevice** recibe una corriente de sonido en su única entrada y la reproduce en el hardware del PC.

**DV_WaveGeneradorDevice** genera un sonido de ciertas características y lo suministra en su única salida (se utiliza principalmente para pruebas).

**EF_Compresión** implementa el efecto de compresión. Recibe una corriente de sonido en su entrada, y eleva la amplitud de las señales más débiles, mientras que las señales fuertes se ven menos afectadas. El denominado *rango dinámico* de una señal describe el rango de amplitud desde la señal más débil de una grabación hasta la señal más fuerte. El resultado de la compresión es, pues, un rango dinámico más reducido, lo cual puede ser necesario debido a limitaciones del dispositivo de grabación empleado (que podrían ocasionar la pérdida de algunos sonidos) o por preferencias personales.

**EF_Distortion** es el componente de distorsión. La amplitud de una señal puede limitarse a ciertos niveles máximos; si la onda original sobrepasa esos niveles por arriba o por abajo (lo que se conoce como *saturación*), la señal se "recorta". En el mundo real esto puede ocurrir

por limitaciones prácticas de la circuitería o de los dispositivos (y en este caso sería un efecto indeseado), pero también se utiliza de manera deliberada en algunos casos (por ejemplo, las guitarras eléctricas se distorsionan frecuentemente, con dispositivos específicos).

**EF_Echo** implementa el efecto de eco. El eco resulta de tomar una señal y sumarla a sí misma con cierto retraso y posiblemente con una amplitud menor. Este componente implementa varios tipos de efectos de eco. El **eco** propiamente dicho añade la señal desplazada repetidamente, pero con una amplitud cada vez menor, lo que tiene el efecto de una repetición cada vez más silenciosa (como el eco natural). El llamado **delay** añade la señal sólo una vez; el efecto es el del mismo sonido reproducido dos veces pero no simultáneamente (aunque el retraso suele ser bastante corto). La **reverberación** intenta emular el efecto del eco que se produce desde diferentes paredes (a diferentes distancias y ángulos) en una habitación, y este efecto se consigue añadiendo la señal con diferentes retrasos y disminuciones de amplitud. EF_Echo es un ejemplo de componente cuyo comportamiento se puede parametrizar en un alto grado.

**EF_Gain** se limita a multiplicar la amplitud de la señal por un factor. Un factor de 2 produce una señal "el doble de fuerte" que la original, y un factor de 0 produciría silencio.

**EF_SepChannels** tiene una entrada y dos salidas. La señal de entrada debe ser estéreo, y los componentes (canales) izquierdo y derecho de dicha señal se separan en dos señales mono.

**EF_JoinChannels** tiene dos entradas y una salida; recibe dos señales mono, y las combina en una señal estéreo.

Se pueden implementar (como de hecho ha ocurrido) otros componentes: puertas de ruido, filtros de frecuencia, mezcladores, etc. Además, WaveX incorporará componentes para procesar señales ya grabadas en disco y grabar su salida también a disco; basta con crear nuevos componentes para ello. El núcleo del sistema se extenderá también para sincronizar diferentes señales, de manera que se pueda utilizar como un pequeño estudio de grabación y mezcla de bajo coste. Para más información sobre el proyecto WaveX, véase [WaveX].

Ya se ha dicho que los componentes WaveX tienen entradas y salidas bien definidas. Su funcionamiento se puede modificar también a través de diversos **parámetros**. Por ejemplo, DV_WaveInDevice se puede configurar para grabar señales a diferentes frecuencias de muestreo (8000, 11025, 22050 y 44100 muestras por segundo), diferente número de canales (1 para señales mono y 2 para señales estéreo), y diferentes longitudes de buffer. DV_Gain puede elevar o reducir la amplitud de una señal dependiendo del factor de ganancia.

Hay una tensión entre los parámetros y las entradas; el papel de los parámetros es permitir la creación de instancias de componente con cierto grado de libertad sobre su comportamiento, sin necesidad de crear entradas adicionales y componentes "constantes" simplemente para introducir meros valores de configuración. Esto añadiría al diseño una complejidad innecesaria.

```
MODULE Input DV_WaveInDevice.DLL
    PARAM DesiredChannels 2
    PARAM DesiredBitsPerSample 16
    PARAM DesiredBufferSize 4096
    PARAM DesiredSamplesPerSecond 44100

MODULE Separation EF_SepChannels.DLL

MODULE Joining EF_JoinChannels.DLL

MODULE Play DV_WaveOutDevice.DLL
    PARAM DesiredChannels 2
    PARAM DesiredBitsPerSample 16
    PARAM DesiredSamplesPerSecond 44100

LINK Input out Separation in
LINK Separation left Joining right
LINK Separation right Joining left
LINK Joining out Play in
```

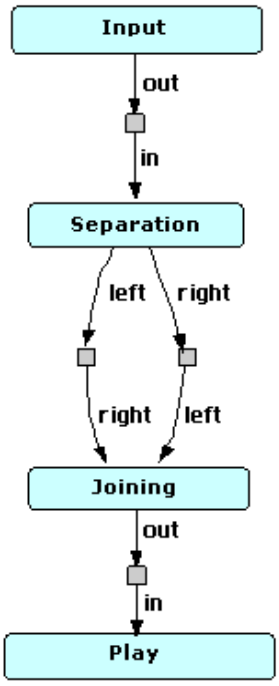



*Ilustración 31. Un guión WaveX que describe un sistema para invertir los canales del estéreo; a la derecha, representación gráfica de este sistema (captura de pantalla del prototipo Itacio-XBD).*

El usuario, pues, creará un fichero de "topología" que describe cierta configuración de componentes. Por ejemplo, el fichero de la Ilustración 31 invierte los canales izquierdo y derecho de una señal estéreo. Las palabras clave necesarias son muy simples: la sentencia MODULE declara una instancia de módulo, indicando qué componente (DLL) lo implementa. Las sentencias PARAM siguen al componente al cual afectan. Las sentencias LINK se refieren a las instancias de componente declaradas previamente, indicando cómo deben interconectarse. La sintaxis se puede deducir fácilmente del siguiente ejemplo:

```
MODULE <Nombre> <DLLComponente>
PARAM <NombreParámetro> <Valor>
LINK <MóduloOrigen> <Salida> <MóduloDestino> <Entrada>
```

El núcleo de WaveX carga e interpreta este guión, creando el sistema funcional mediante la combinación de los componentes necesarios (véase la Ilustración 32).

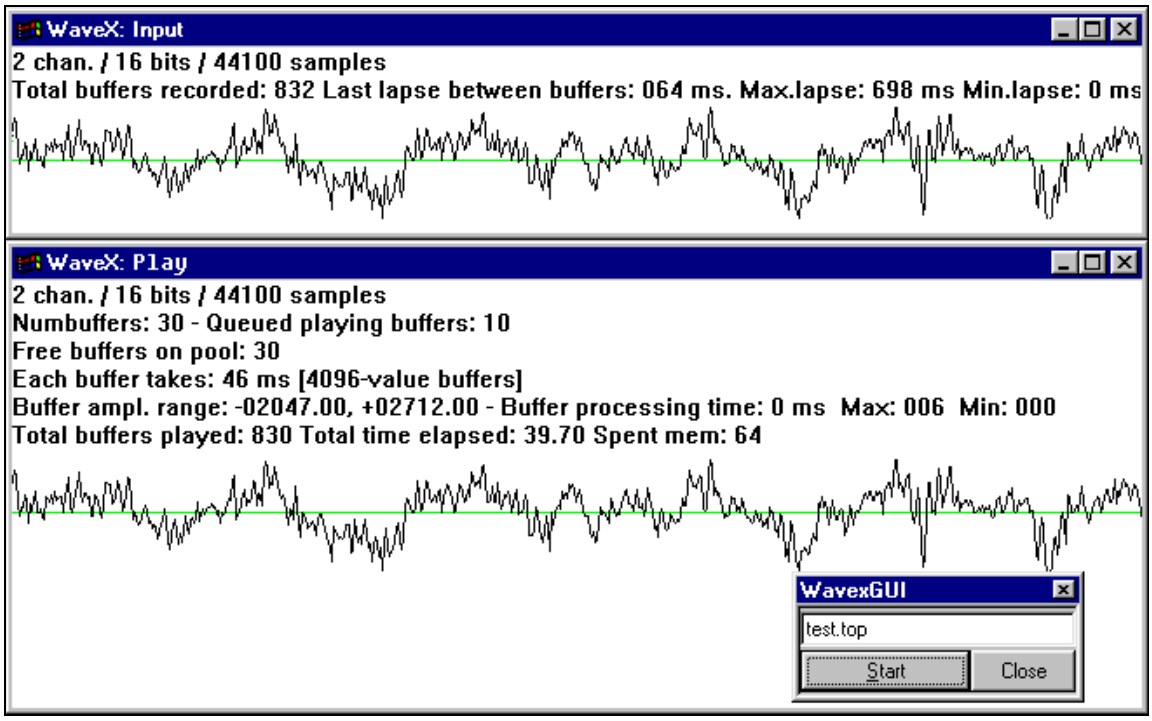



*Ilustración 32. El sistema de ejemplo de la Ilustración 31 funcionando en WaveX: el núcleo (WavexGUI), el componente Input y el componente Play. Los componentes Separation y Joining no tienen representación visual.*

En este ejemplo tan simple, puede parecer innecesaria la verificación. Pero incluso en este caso se pueden cometer errores. Por ejemplo, si el dispositivo Input se configura para grabar una señal mono (`PARAM DesiredChannels 1`) el sistema no funcionará. El componente Separation necesita una señal estéreo, mientras que Input estaría configurado para producir una señal mono. Hay una dificultad especial, y es que los problemas pueden surgir en componentes distantes; por ejemplo, si Input estuviera configurado para grabar a 44100 muestras por segundo pero Play estuviera configurado para reproducir a 22050 muestras por segundo, el problema se manifestaría en la conexión entre Play y Joining (en realidad no hay ningún problema hasta ese punto, porque los componentes intermedios pueden manejar cualquier frecuencia de muestreo). Algunos dispositivos pueden requerir un margen de amplitud limitado (por ejemplo, el hardware de reproducción puede incluso resultar dañado físicamente por una señal demasiado fuerte, por lo que sería apropiado poder establecer límites en alguna parte). Por supuesto, WaveX está diseñado para soportar sistemas mucho más complejos (que involucran más componentes y más parámetros interrelacionados) que el pequeño ejemplo de la Ilustración 31, y existen muchas malformaciones en potencia.

Hay aún un problema importante cuando se utiliza WaveX en tiempo real. Ya se ha mencionado que cada buffer contiene cierto lapso de sonido; su duración depende del tamaño de buffer, frecuencia de muestreo, resolución y número de canales de la señal. El buffer generado por un componente de grabación pasa de un componente a otro, siendo procesado de diversas maneras, y habitualmente termina en un componente de reproducción. La ventana temporal disponible para hacer esto es la duración de un buffer; si el tiempo necesario para procesar por completo un buffer es mayor que el lapso de tiempo

de reproducción / grabación que el buffer contiene, el sistema no estará listo para procesar de manera inmediata el siguiente buffer grabado, puesto que el dispositivo grabador produce buffers a un ritmo constante e ininterrumpido.

Por supuesto, se puede intentar afrontar estos problemas de conexión con pre/postcondiciones o aserciones "tradicionales" [Me99]. Pero esto tiene ciertas desventajas:

- La descripción de la interfaz (y el conocimiento sobre el uso esperado) de un componente estará enterrada en el código fuente de procesamiento, mezclada con él.
- El proceso de gestionar una incongruencia durante la ejecución no termina simplemente detectándola; debe informarse de la presencia de un error, describirlo y manejarlo adecuadamente, y la gestión coherente de excepciones puede no ser una tarea fácil (especialmente cuando están involucrados componentes independientes, como es el caso).
- La verificación basada en aserciones muestra incongruencias sólo si las aserciones se violan durante la ejecución; hasta cierto punto, son equivalentes a las pruebas.
- En general (al menos en las herramientas de desarrollo más extendidas y en los hábitos de la mayoría de los programadores) las aserciones de lenguajes imperativos no se analizan estáticamente. Deben ejecutarse; el sistema debe realmente construirse y probarse, en lugar de verificar el diseño por adelantado.

**Aplicación de Itacio a WaveX**

Puesto que WaveX es un procesador de sonido muy versátil, y su usuario ha de combinar múltiples componentes preexistentes de muchas formas distintas, se puede esperar que el diseño de procesadores de sonido sea una tarea muy propicia para cometer errores; además, la versión final de WaveX tendrá muchos más componentes, y posiblemente algunos de ellos funcionen sólo con ciertos tipos de señales, por lo que el usuario también tendrá que introducir adaptadores en su diseño, elevando la complejidad. Es deseable ayudar al diseñador a detectar problemas potenciales tan pronto como sea posible (en tiempo de diseño); asimismo, un sistema que pueda señalar directamente las inconsistencias y explicarlas sería muy útil.

Por estas razones, Itacio entra en juego. En lugar de recargar los diversos componentes con verificaciones C++ (y manejar las diferentes posibilidades de error), se puede usar Itacio para describir los componentes WaveX y veridicar cada diseño antes de construir o ejecutar realmente el sistema. Este es el proceso que se va a describir a continuación. Hay que hacer notar que, a menos que se diga explícitamente, el prototipo de Itacio utilizado con WaveX es exactamente el original (Itacio-XDB), y no se han añadido capacidades específicas para WaveX.

Antes de realizar cualquier uso de Itacio, debemos abordar la **instanciación** del modelo, definiendo los conceptos de componente, fuente, sumidero, etc. en el dominio del problema. En este caso, parece obvio que cada componente WaveX, con sus entradas y salidas, puede ser identificado directamente como un componente Itacio con sus sumideros y fuentes, respectivamente.

Con esta estrategia en mente, la primera tarea es introducir en Itacio la definición estructural de todos los posibles componentes: su nombre, fuentes, y sumideros. Una vez que se ha hecho esto, se pueden usar sentencias de programación lógica para poner por escrito los requisitos y garantías de cada componente de manera separada. Como ejemplo, la Ilustración 33 muestra la frontera del componente DV_WaveInDevice; esta definición es una "plantilla" del componente, y se crearán instancias del mismo en cada sistema cuando sea necesario. En este caso, el componente no tiene ningún sumidero. Tiene sólo una fuente (llamada *out*), y ofrece (en el momento en que se realizó la captura de pantalla) cuatro garantías.

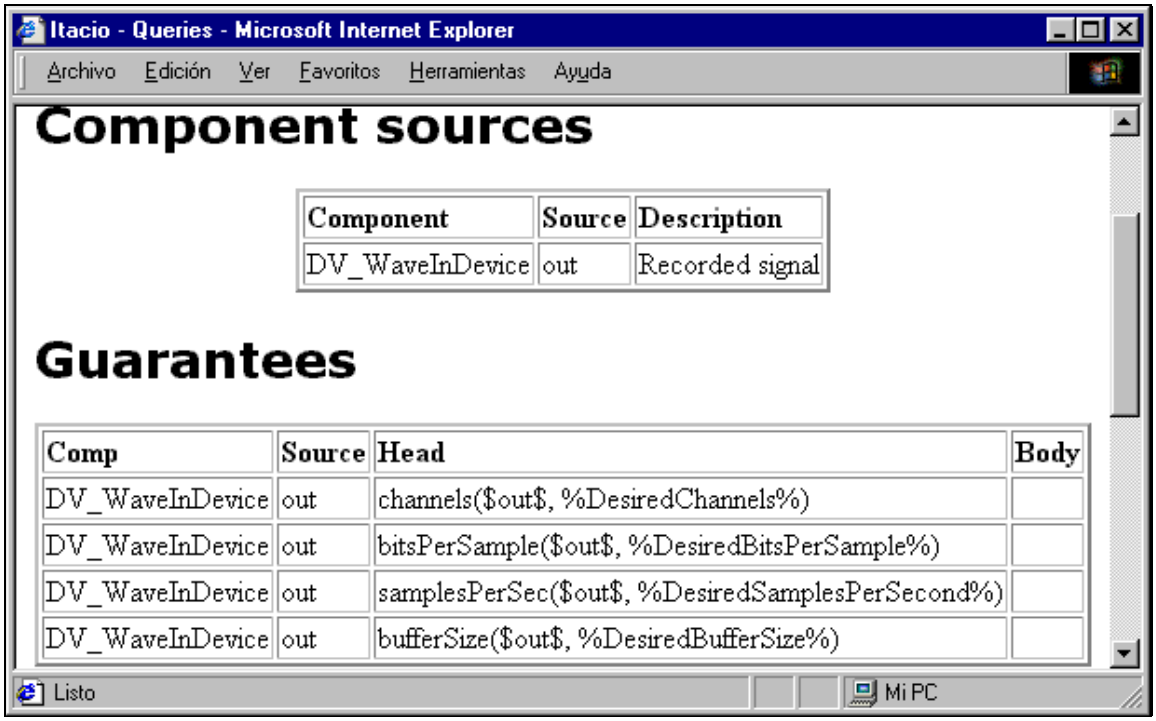



*Ilustración 33. Frontera de DV_WaveInDevice (captura de pantalla del prototipo Itacio-XDB).*

Puede verse que las garantías son, de hecho, código Prolog bajo la forma de cláusulas Horn; en este caso, ninguna de las reglas tiene cuerpo, por lo que todas ellas son hechos. El código Prolog aquí utilizado sigue un par de normas: los nombres de átomo encerrados entre signos **$** deben hacer referencia a fuentes o sumideros del componente, y se sustituirán automáticamente por los átomos que representan las conexiones correspondientes (como se ha descrito en el capítulo 4). Los nombres de átomo encerrados entre signos **%** representan parámetros, y serán también sustituidos por los valores concretos de cada instancia de componente. En este caso, DV_WaveInDevice está garantizando que la señal que produce tendrá el número de canales, resolución, frecuencia de muestreo y tamaño de buffer que su configuración le marca.

Se introduciría información análoga en Itacio para todos los componentes que puedan utilizarse en la construcción de un sistema (en este caso, el "sistema" será un procesador de sonido concreto). En la Ilustración 34 (representada en forma textual y no como captura de pantalla por razones de espacio) aparece un segundo ejemplo, la descripción de EF_SepChannels. Puede verse que este componente requiere que su señal de entrada sea

estéreo, y se compromete a generar una señal mono en su fuente izquierda en la que todas las demás características permanecen sin cambios; lo mismo para la fuente derecha.

| Componente: EF_SepChannels | |
|---|---|
| **Sumideros**: in | **Fuentes**: left, right |

**Requisitos**:

```
channels($in$, 2).
```

**Garantías**:

```
channels($left$, 1).
bitsPerSample($left$, X) :- bitsPerSample($in$, X).
samplesPerSec($left$, X) :- samplesPerSec($in$, X).
bufferSize($left$, X) :- bufferSize($in$, X).

channels($right$, 1).
bitsPerSample($right$, X) :- bitsPerSample($in$, X).
samplesPerSec($right$, X) :- samplesPerSec($in$, X).
bufferSize($right$, X) :- bufferSize($in$, X).
```

*Ilustración 34. Frontera de EF_SepChannels.*

Una vez que están definidas las plantillas de componente, los pasos finales son los de la creación de un sistema concreto. Basta con definir instancias de los componentes existentes y conectar sus fuentes y sumideros como sea necesario; también hay que dar a los parámetros de cada instancia de componente los valores adecuados. Para estas tareas, la versión actual de Itacio-XDB presenta una interfaz muy pobre, orientada a la propia base de datos, pero por supuesto sería posible desarrollar un buen editor gráfico más fácil de usar.

Siguiendo el procedimiento descrito, se ha definido un sistema de ejemplo. Aunque Itacio no ofrece aún utilidades de *edición* gráfica, es capaz de *mostrar* una representación gráfica de cualquier sistema, en formato web (mediante la combinación de XML y VML descrita en el capítulo 5.1.2) y en formato PostScript (según lo descrito en 5.1.3). Además de que la representación gráfica ayuda a una mejor comprensión del sistema, el aspecto más interesante de la misma es que también incorpora los resultados del proceso de verificación.

En 4.2.4 ya se ha descrito cómo Itacio puede generar la base de conocimientos que representa el sistema e interactuar con el motor de inferencia de ECLiPSe para obtener información sobre la corrección de cada una de las conexiones. Si una conexión no es correcta, el grafo lo mostrará con claridad; el usuario puede entonces hacer clic sobre la conexión en cuestión y obtener una explicación sobre el fallo. La  representa el caso en que el componente Input ha sido configurado para generar una señal mono de 22050 Hz mientras que el componente Play espera una señal de 44100 Hz y el componente Separation necesita una señal estéreo. El sistema marca la conexión errónea con un cuadrado rojo de mayor tamaño, y la pulsación del ratón sobre él muestra las explicaciones que permiten al usuario corregir su diseño.

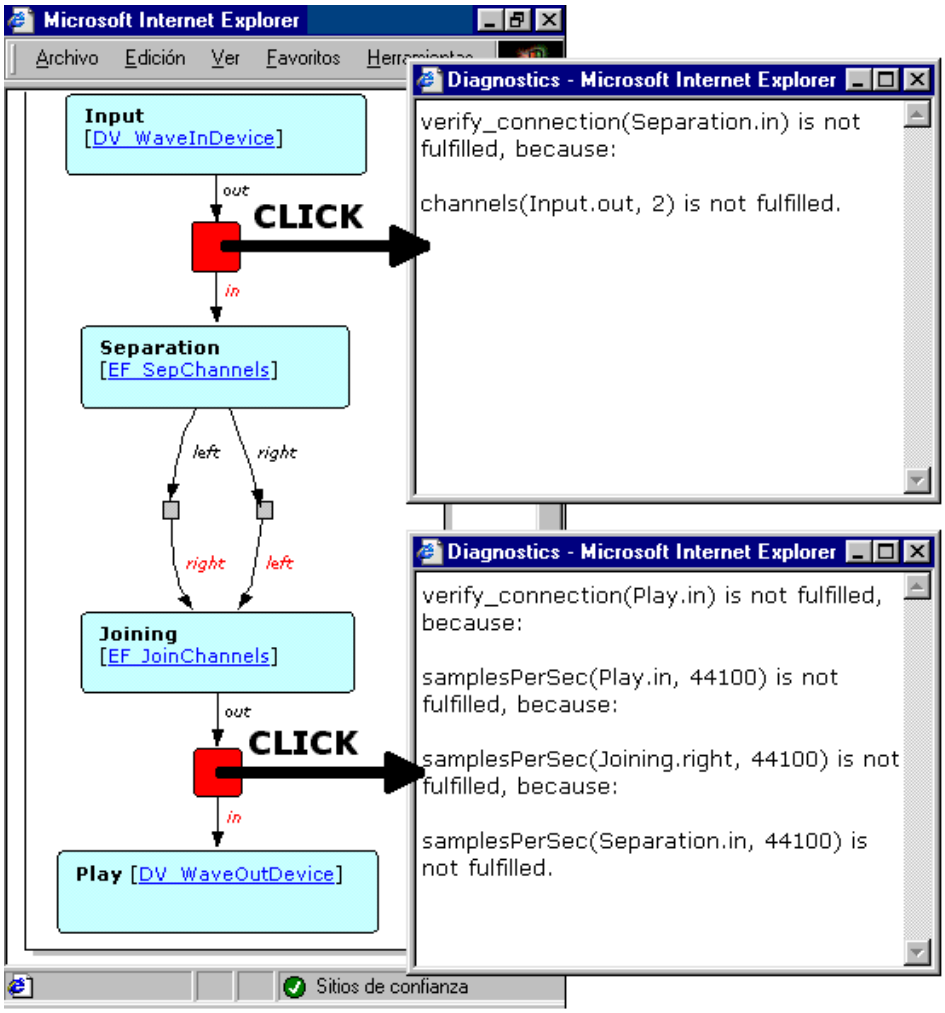



*Ilustración 35. Un sistema con errores y sus explicaciones.*

Por lo que se refiere al problema del procesamiento en tiempo real, se pueden añadir fácilmente las garantías y requisitos para tener en cuenta este problema. Basta con añadir a los requisitos de DV_WaveOutDevice lo siguiente:

```
buffer_milliseconds($in$, BUFFER_TIME),
buffer_processing_time($in$, PROCESSING_TIME),
PROCESSING_TIME < BUFFER_TIME
```

Puede verse que es necesario un predicado auxiliar con el fin de calcular la duración de un buffer a partir de su tamaño y demás datos, y que esto no está ligado a ningún componente particular. Es conocimiento general, por lo que se incluirá en la biblioteca del sistema (la ***L*** mencionada en el capítulo 4) en forma de una simple regla:

```
buffer_milliseconds(SIGNAL, TIME) :- channels(SIGNAL, CHANNELS),
  samplesPerSec(SIGNAL, SAMPLES_PER_SECOND),
  bufferSize(SIGNAL, BUFSIZE),
  TIME is ((1000 * BUFSIZE) / CHANNELS) / SAMPLES_PER_SECOND.
```

Además, cada componente incorporará su propia información sobre rendimiento. En este caso, hemos incluido sólo el tiempo de procesamiento en el caso peor, expresado en

milisegundos, y la información puede obtenerse empíricamente o calcularse a partir de la complejidad del algoritmo involucrado. En este caso, EF_SepChannels podría arrojar un tiempo de procesamiento en el caso peor de unos 10 ms:

```
buffer_processing_time($left$, X) :-
  buffer_processing_time($in$, TIME_INPUT), X is 10 +
  TIME_INPUT.

buffer_processing_time($right$, X) :-
  buffer_processing_time($in$, TIME_INPUT), X is 10 +
  TIME_INPUT.
```

En el caso de EF_JoinChannels, puesto que su propio tiempo de procesamiento nunca excede los 10 ms, el comportamiento de este componente le obligaría a esperar por las dos entradas (izquierda y derecha) por lo que el tiempo de procesamiento acumulado sería el peor de los dos canales añadiendo sus propios 10 ms:

```
buffer_processing_time($out$, X) :-
  buffer_processing_time($left$, LEFT),
  buffer_processing_time($right$, RIGHT), LEFT > RIGHT,
  X is LEFT + 10.

buffer_processing_time($out$, X) buffer_processing_time($left$,
  LEFT), buffer_processing_time($right$, RIGHT), LEFT =< RIGHT,
  X is RIGHT + 10.
```

Al ser este ejemplo deliberadamente simple, no aparecen realmente sentencias de Programación Lógica con Restricciones en la descripción declarativa; sólo hay sentencias de programación lógica sin más. Estamos utilizando sólo tiempos de proceso en el caso peor, y una simple comparación basta para verificar que la ventana temporal disponible no se desborda. Pero si fuese necesario un análisis más profundo del comportamiento del sistema, se podría utilizar información sobre tiempos de procesamiento en el caso mejor y en el caso peor (es decir, rangos), y esto sí requeriría el uso de Programación Lógica con Restricciones, puesto que el Prolog tradicional no es lo más indicado para unificar o comparar dominios y rangos.

Al introducir estos requisitos y garantías en Itacio, se obtiene una descripción más detallada del comportamiento de los componentes. En el ejemplo que se está manejando, el sistema sigue siendo válido, porque el tiempo total de procesamiento en el caso peor cabe holgadamente en la ventana temporal. Un buffer de 8192 muestras para una señal estéreo de una frecuencia de muestreo de 44100 muestras por segundo puede contener 92 ms, y el tiempo de proceso en el caso peor ronda los 20 ms con los datos de eficiencia que hemos manejado (y estos datos son, deliberadamente, muy pesimistas). Pero si se añadiese un componente más complejo, las cosas podrían cambiar. Como ejemplo, se ha definido un componente a efectos de prueba (llamado EF_CostlyOperation), que simplemente "finge" requerir un tiempo de procesamiento grande. Si se introduce este componente como una transformación adicional entre la salida `right` de EF_Separation y la entrada `left` de EF_Joining, no ocurrirá nada si se configura EF_CostlyOperation para que emplee 5 ms de tiempo en el proceso; pero si se configura para consumir 100 ms, la ventana de tiempo se excede, y el sistema detecta este problema de la forma habitual (Ilustración 36).

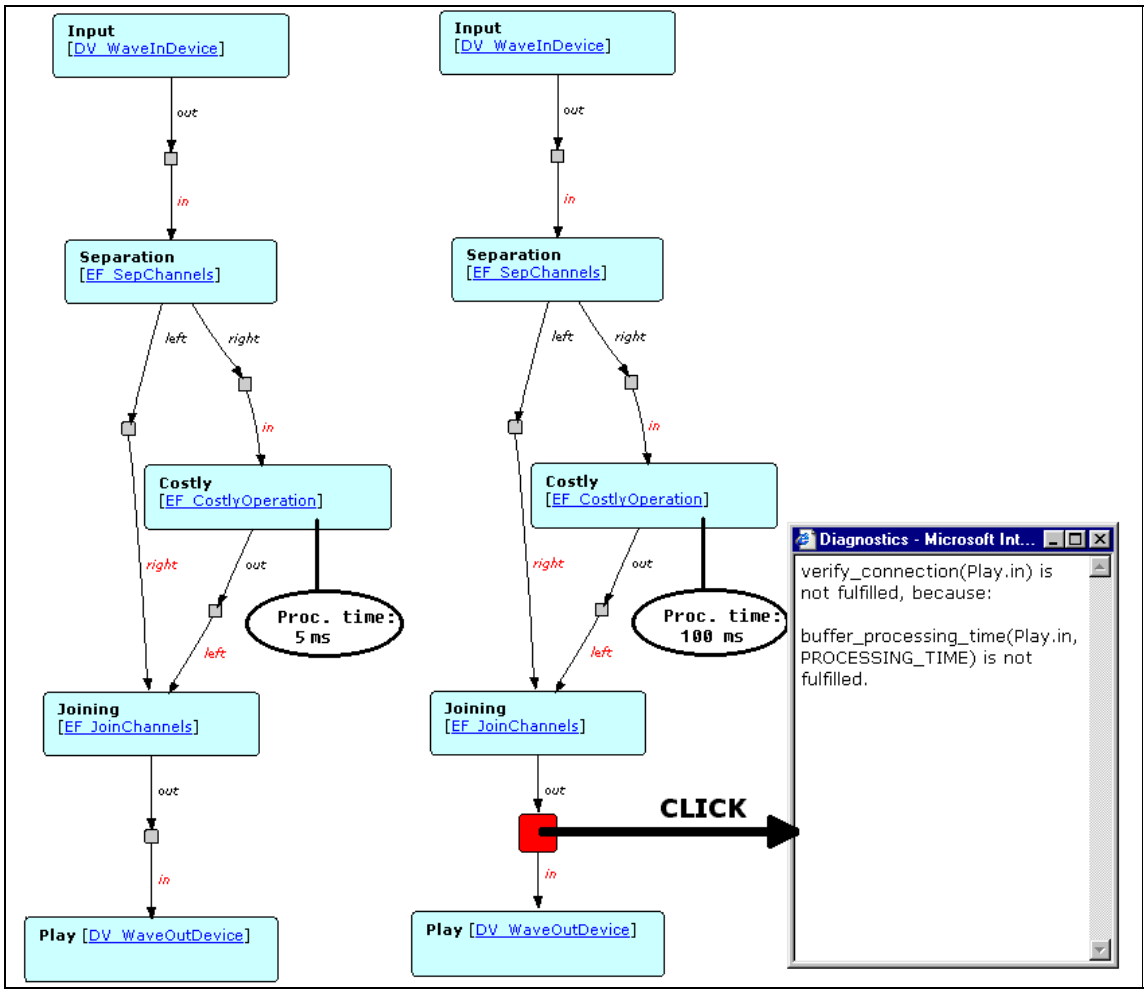



*Ilustración 36. El sistema de ejemplo con restricciones de eficiencia. A la izquierda, un sistema correcto; a la derecha, un sistema cuyo tiempo de proceso desborda la ventana temporal de grabación.*

### Extensión de Itacio-XDB específica para WaveX

Cabe hacer un comentario final respecto al proceso de desarrollo para módulos WaveX. Tras comprobar experimentalmente la hipótesis de que el uso de Itacio para verificar sistemas WaveX era adecuado, se añadió al prototipo Itacio-XDB funcionalidad específica: la capacidad de generar el fichero textual de descripción de topología (véase Ilustración 31, pág. 156) a partir de la información almacenada en Itacio-XDB. Este es un proceso elemental de extracción y traducción de la información presente en una base de datos, pero permite al usuario de WaveX utilizar Itacio como un entorno de desarrollo semi-gráfico; una vez que el diseño es correcto y válido de acuerdo con todo el conocimiento disponible sobre los componentes utilizados, el usuario puede ver directamente el sistema en funcionamiento. Esta estrecha integración entre la herramienta de verificación (Itacio) y la herramienta de desarrollo / ensamblaje (WaveX en este caso) junto con una buena implementación de la herramienta de verificación (que en lugar de ser un prototipo sería un sistema en producción con interfaces de usuario adecuadas y editores gráficos) es,

seguramente, la clave para incorporar Itacio de manera efectiva en cualquier proceso de desarrollo basado en componentes.

**Conclusiones**

El mecanismo de verificación de signaturas que ofrecen hoy en día modelos de componentes como COM, CORBA o JavaBeans parece no proteger a los desarrolladores y usuarios de las combinaciones incorrectas de componentes. El modelo de componentes Itacio se gestó con esta motivación en mente. Este apartado describe su aplicación a un caso típico de desarrollo basado en componentes (WaveX). Aunque los ejemplos presentados aquí son deliberadamente simples por razones de espacio y legibilidad, muestran que la generalidad de la programación lógica con restricciones es un gran beneficio para describir todo tipo de restricciones de uso de los componentes, que seguramente no podrían ser verificadas de forma estática y automática en las plataformas de componentes actualmente extendidas a nivel comercial.

Se puede argumentar que el enfoque de Itacio obligaría al desarrollador a aprender un "segundo lenguaje". Es, desde luego, cierto que recoger y utilizar conocimiento en el proceso de diseño y/o desarrollo no sale gratis; pero creemos que la propuesta de Itacio es claramente viable. La aplicación efectiva de cualquier otra técnica de especificación, incluyendo muchos métodos formales, requeriría también un esfuerzo extra. Itacio se basa en programación (declarativa) y en un mecanismo de inferencia bien conocido, y estos factores pueden ser de gran valor para su adopción en una organización de desarrollo.

El sistema WaveX no es, de por sí, el objetivo de esta tesis, pero ilustra claramente que el modelo Itacio podría jugar un importante papel para ayudar al usuario final de este producto a diseñar procesadores de sonido. En este caso se ha utilizado el prototipo general de Itacio, pero en versiones futuras de WaveX el sistema de verificación podría estar *incorporado* al producto final, mejorando en gran medida la facilidad de uso del mismo y la satisfacción del usuario (al evitarse errores de ejecución). En general, la integración del modelo en las herramientas de desarrollo de cada caso particular, que parece perfectamente posible desde el punto de vista técnico, sería un gran paso adelante en la adopción del modelo Itacio .

## 5.2.5. Modelo de fiabilidad de Hamlet et al

Las plataformas de componentes habitualmente utilizadas (véase capítulo 2.9) permiten describir los componentes desde el punto de vista de su interfaz (limitando este concepto a descripciones de signaturas) y verificar la combinación de dichas interfaces. No obstante, es frecuente que se plantee la necesidad de pensar en otras características de los componentes, como pueden ser restricciones de eficiencia o de fiabilidad (entendiendo la fiabilidad como la tasa de fallos).

Por lo que se refiere a la fiabilidad, lo cierto es que el software se usa cada vez más en sistemas en los que las consecuencias de un fallo son graves (en un plano médico, militar, medioambiental, económico, etc.) Además, con objeto de abaratar costes va en aumento la tendencia a utilizar en el desarrollo de aplicaciones componentes preexistentes adquiridos a terceros. En ocasiones, se trata de aplicaciones críticas. Lógicamente, esto plantea la necesidad de que la composición de componentes software pueda realizarse bajo condiciones de fiabilidad conocidas y predecibles [PI98].

En este marco se puede encuadrar el trabajo de Hamlet et al [HMW01]. Estos autores presentan una teoría de la fiabilidad basada en componentes; en este capítulo se propone una representación de dicha teoría en el modelo Itacio, lo que constituye un caso más de aplicación de Itacio en un campo diferente (el modelado de la fiabilidad de un sistema). En primer lugar se describirá de forma resumida el modelo presentado en [HMW01], y después se detallará cómo se ha aplicado a su vez el modelo Itacio a este sistema, utilizando el mismo ejemplo de Hamlet et al. En este caso, la aplicación de Itacio debe entenderse como un ejercicio demostrativo y no como un caso real de uso, puesto que la propia teoría de Hamlet et al es, por el momento, sólo teoría, y hasta donde nos consta este trabajo se encuentra aún bajo desarrollo y no ha finalizado.

**Una teoría de la fiabilidad de un sistema basada en componentes**

La mayor parte de la investigación en software basado en componentes se ha dirigido a la especificación funcional de componentes. Otro aspecto igualmente importante es la calidad de los mismos; de ahí la propuesta de Hamlet de una teoría básica de la fiabilidad basada en componentes. De lo que se trata es de que los desarrolladores de los componentes puedan realizar mediciones, que son utilizadas después por quienes diseñan sistemas basados en esos componentes para calcular (sin implementación ni pruebas) la fiabilidad del sistema resultante.

En este caso, se trata –en palabras de sus autores- de una teoría *microscópica* que describe con detalle cómo las propiedades de los componentes se reflejan en sistemas que hayan sido diseñados usando dichos componentes. En el marco de esta teoría fundacional, pueden verse los componentes simplemente como pequeñas subrutinas.

En la ingeniería eléctrica o mecánica (que ya han sido citadas en otros puntos de esta tesis) los componentes se describen en un manual, que incluye una "hoja de datos" para cada componente. Dicha hoja de datos describe lo que hace un componente, y también (y es un aspecto importante) establece restricciones que permiten al diseñador del sistema tomar la decisión de si un componente es apropiado o no (lo suficientemente bueno) para la aplicación; por ejemplo, los componentes mecánicos incluyen información sobre su tiempo de vida previsto, y esta información puede estar detallada para diferentes rangos de

temperatura posibles en los que se vaya a utilizar dicho componente. Basándose en las hojas de datos, un diseñador de sistemas puede extraer conclusiones sobre el comportamiento futuro del sistema que está construyendo.

En el caso de componentes software, sería deseable tener información similar en la que basar ese tipo de predicciones. Especialmente si se usan componentes de terceros, el no disponer de ningún tipo de información al respecto deja al diseñador en una difícil disyuntiva: si se decide a construir el componente a medida, con las dificultades que conlleva el estimar su fiabilidad, o a comprar el componente ya hecho, con las dificultades (aún mayores) que conlleva estimar la fiabilidad del trabajo que otros han hecho.

La teoría de Hamlet et al pretende ofrecer una base para la descripción de componentes software mediante hojas de datos y el uso de esos datos para calcular la fiabilidad de un sistema formado por estos componentes.

**Perfil operacional**

El comportamiento de un componente software puede verse afectado en gran medida por el entorno en el que se utiliza. Las pruebas de un componente se realizan en cierto entorno, mientras que el uso final de ese componente puede tener lugar en condiciones muy diferentes. Por ejemplo, un componente probado con valores de entrada elegidos según una distribución uniforme puede arrojar cierta tasa de fallos aparentemente aceptable, pero si ese componente se utiliza con valores de entrada que no siguen una distribución uniforme, sino que se concentran en un rango especialmente conflictivo, la tasa de fallos en producción puede variar drásticamente.

Para Hamlet et al esto denota claramente la necesidad de un **perfil operacional** que describa la situación en la que se llevaron a cabo las pruebas y también la situación en la que el componente se utilizará. La teoría que nos ocupa se apoya en la propuesta de que la información que debe aparecer en la hoja de datos se referirá a la fiabilidad estadística (la información sobre el comportamiento del componente desde un punto de vista funcional es igualmente importante, pero no es objeto de dicha teoría). Las ideas básicas en las que se asienta la teoría son:

**Correspondencia perfil / fiabilidad**. Los perfiles operacionales deben tenerse en cuenta cuando se miden los parámetros de un componente para crear una hoja de datos. Puesto que el desarrollador del componente no sabe cómo se utilizará este, ni qué perfil tendrá que afrontar cuando sea utilizado, la información de la hoja de datos incluirá como parámetro el perfil en cuestión. Es decir, la hoja de datos especifica correspondencias entre perfiles y parámetros de fiabilidad.

**Subdominios del componente**. Un componente tiene una partición natural de su espacio de entrada en subdominios funcionales, y la descripción práctica de su perfil operacional es un vector de pesos sobre dichos subdominios. Esta forma de perfil operacional permite al desarrollador probar un componente sin conocer dicho perfil, y los pesos concretos del perfil de uso pueden aplicarse posteriormente.

De acuerdo con estos dos conceptos básicos, la fabricación y uso de un componente incluiría las siguientes etapas:

- El desarrollador del componente define los subdominios "naturales" del espacio de entrada de un componente.
- El desarrollador mide la fiabilidad del componente en cada subdominio.
- El desarrollador publica la hoja de datos que incluye los subdominios, los datos de fiabilidad y los *pesos* de cada subdominio durante las pruebas.
- El diseñador de un sistema, usando las hojas de datos, calcula los datos de fiabilidad para diferentes combinaciones de componentes y puede decidir qué componentes satisfacen sus requisitos.
- En caso de no conseguir la fiabilidad deseada, el diseñador puede elegir componentes alternativos y / o cambiar la estructura del sistema.

**Un ejemplo de combinación de perfiles operacionales**

En [HMW01] se ofrece un ejemplo básico que ilustra el uso de este modelo. Supóngase que se dispone de dos componentes A y B, componentes que realizan algún tipo de cálculo u operación en secuencia, de modo que la salida de A constituye la entrada de B. Este ejemplo permite mostrar en la práctica las ideas básicas del perfil operacional, y también la resolución de un problema básico que esta teoría tiene que afrontar: la concatenación de componentes, o *composición secuencial*.

Como ya se ha explicado, el componente A vendrá acompañado de una hoja de datos que ofrezca información sobre su fiabilidad (dicho de otro modo, sobre su tasa de fallos). Pero según lo expuesto, este componente no tendrá siempre la misma fiabilidad; dependerá de la situación en la que se utilice, es decir, del perfil operacional.

Un primer paso de la creación de la hoja de datos consiste en identificar los subdominios significativos en el espacio de entrada de este componente. Téngase en cuenta que el número de datos de entrada del componente es indiferente; un conjunto de valores siempre puede considerarse como un solo valor vectorial. Simplemente, en vez de manejar un solo valor se tendrá un espacio de datos de entrada n-dimensional, y la identificación de los subdominios puede resultar más o menos difícil, pero ese problema no es el objetivo de este ejemplo.

Supóngase, entonces, que para el componente A se identifican tres subdominios significativos, $A_1$, $A_2$ y $A_3$. A tiene unas tasas de fallo, determinadas experimentalmente; para valores del primer subdominio, A tiene una tasa de fallos de 0.01, para valores del segundo subdominio la tasa de fallos es 0, y para valores del tercer subdominio la tasa es 0.001. El vector de tasas de fallo sería, pues,

$$F_A = <f_{A1}, f_{A2}, f_{A3}> = <0.01, 0.0, 0.001>.$$

Análogamente, se tiene un componente B, para el cual se identifican cuatro subdominios. Se determina experimentalmente (con pruebas aleatorias sobre cada subdominio) que el vector de tasas de fallo es

$$F_B = <f_{B1}, f_{B2}, f_{B3}, f_{B4}> = <0.1, 0.0, 0.0, 0.02>.$$

Con esta información se tienen hojas básicas de datos que describen individualmente a los componentes A y B respecto a su fiabilidad; pero lógicamente para llegar a datos concretos se necesitan datos sobre el perfil operacional en el que se utilizará el componente.

Para el caso de A, la existencia de tres subdominios nos lleva a expresar el perfil operacional como un vector de tres probabilidades, que indica el porcentaje de valores de entrada de cada subdominio que se espera se presente a la entrada de A en las circunstancias concretas de uso. Por ejemplo, supóngase que el componente A se va a utilizar en un entorno en el que el 30% de los valores de entrada pertenecen al primer subdominio, el 10% al segundo y el 60% al tercero. El perfil de entrada de A será, pues,

$$Q_A = <k_{A1}, k_{A2}, k_{A3}> = <0.3, 0.1, 0.6>.$$

Esa información es suficiente para conocer la tasa de fallos del componente A, ya que se conoce su perfil operacional y la tasa de fallos de cada subdominio. La fiabilidad[5] (*reliability*) de A bajo ese perfil de operación será, pues:

$$R_A = \Sigma\, k_{Ai} \times (1 - f_{Ai}) = 0.3 \times (1 - 0.01) + 0.1 \times (1 - 0.0) + 0.6 \times (1 - 0.001) \cong \mathbf{0.996}$$

En esas circunstancias de uso (es decir, bajo ese perfil operacional) el componente A funcionaría correctamente el 99,6% de las veces.

Para calcular la fiabilidad del componente B, el proceso es el mismo, pero aquí aparece un problema adicional. El perfil operacional de entrada de B, $Q_B$, no puede establecerse arbitrariamente, ya que las entradas de B son las salidas de A; por tanto, $Q_B$ es función de la salida de A, que a su vez es función de su entrada. Lo que se necesita es una **correspondencia de transformación de perfiles** para A, que tendrá que figurar en su hoja de datos o bien ser deducido experimentalmente con un muestreo uniforme. En la Ilustración 37 se recopilan los datos de ambos componentes. Por ejemplo, según esa tabla, todos los valores de entrada del subdominio $A_2$ producen una salida en A que se encuadra en el subdominio $B_2$ de B. Si la entrada de A es del subdominio $A_3$, el 0,2% de las salidas se encuadran en el subdominio $B_2$ de B, el 16,2% de las salidas son del subdominio $B_3$ y el 83,6% del subdominio $B_4$. También puede verse que en este caso a B nunca se le suministra un valor del subdominio $B_1$ (dicho de otro modo, el componente A nunca produce valores de ese subdominio).

**Hoja de datos de A**

Número de subdominios: 3

Tasas de fallos: <0.01, 0.0, 0.001>

Transformación de perfiles:

| Subdom. | De $A_1$ | De $A_2$ | De $A_3$ |
|---|---|---|---|
| $B_1$ | 0 | 0 | 0 |
| $B_2$ | 0.003 | 1.0 | 0.002 |
| $B_3$ | 0.147 | 0 | 0.162 |
| $B_4$ | 0.850 | 0 | 0.836 |

**Hoja de datos de B**

Número de subdominios: 4

Tasas de fallos: <0.1, 0.0, 0.0, 0.02>

Transformación de perfiles:

*[Aquí figuraría la transformación de perfiles adecuada al número de subdominios del siguiente componente.]*

*Ilustración 37. Ejemplos de hojas de datos para los componentes A y B.*

[5] Nótese que aquí se habla de *fiabilidad* (*reliability*) y no de *tasa de fallos*; al tratarse de probabilidades, puede decirse que fiabilidad = 1 - tasa de fallos.

Con la tabla de transformación de perfiles de la Ilustración 37 y dado el perfil operacional de A, $Q_A$, se puede calcular $Q_B$:

$$k_{B1} = 0.3 \times 0 + 0.1 \times 0 + 0.6 \times 0 = 0$$

$$k_{B2} = 0.3 \times 0.003 + 0.1 \times 1.0 + 0.6 \times 0.002 = 0.102$$

$$k_{B3} = 0.3 \times 0.147 + 0.1 \times 0 + 0.6 \times 0.162 = 0.141$$

$$k_{B4} = 0.3 \times 0.850 + 0.1 \times 0 + 0.6 \times 0.836 = 0.757$$

Por tanto el perfil operacional de B en estas circunstancias es:

$$Q_B = <0, 0.102, 0.141, 0.757>$$

Y la fiabilidad de B es:

$$R_B = \Sigma\, k_{Bi} \times (1 - f_{Bi}) = 0 \times (1 - 0.1) + 0.102 \times (1 - 0.0) + 0.141 \times (1 - 0.0) + 0.757 \times (1 - 0.02) \cong \mathbf{0.985}$$

**Aplicación de Itacio al modelo de fiabilidad**

Supóngase que se quiere aplicar el modelo Itacio para describir y verificar restricciones de fiabilidad de los componentes. En este apartado se aplicará el prototipo Itacio-XDB al mismo ejemplo propuesto por Hamlet et al, presentado en el apartado anterior.

El primer paso, como de costumbre, es la instanciación del modelo en el dominio del problema. La asimilación del concepto de componente de Itacio parece clara, ya que en el modelo de Hamlet et al también se hace referencia directa a componentes software; en este caso, la correspondencia es directa.

Respecto a la frontera de los componentes (sus fuentes y sumideros) hay varias opciones, de las que se pueden mencionar sobre todo dos:

a. Los sumideros y fuentes representan las entradas y salidas del componente de forma "directa", es decir, cada sumidero sería un parámetro de entrada y cada fuente un parámetro de salida o valor de retorno. La información sobre perfiles (subdominios, probabilidades) se adjuntaría a cada una de estas entradas y salidas.

b. Los sumideros y fuentes representan a los subdominios de entrada y salida, es decir, cada sumidero representa un subdominio de entrada y cada fuente un subdominio de salida.

Ambos enfoques tienen sus ventajas e inconvenientes. Por ejemplo, la opción b trae como consecuencia que la estructura de un componente representado en Itacio podría depender de a qué componente va a conectarse; los subdominios de salida son algo que depende de los "intereses" particulares del componente al cual van a servir de entrada.

Por su parte, la opción a tiene la ventaja de que la representación del componente sería "fiel" a la imagen habitual de las entradas y salidas del mismo y no dependería de a qué otro componente se conectase, pero en contrapartida no resulta de especial interés en el modelo que nos ocupa. Recuérdese que, de cara a establecer la fiabilidad, el número y tipo de las entradas no es la información esencial; lo importante es la partición que se establece en el dominio de entrada. A estos efectos, es indiferente si se trata de una sola entrada con valores complejos o de muchas entradas con valores simples; la determinación de los subdominios puede resultar más o menos difícil (en un espacio de valores de más o menos

dimensiones), pero una vez determinados dichos subdominios, la fiabilidad del componente no guarda mayor relación con el número y tipo de las entradas.

De cara a representar el ejemplo, se ha optado por la opción b. Por tanto, los componentes aparecerán representados no desde un punto de vista estructural de los parámetros y valores de retorno, sino desde el punto de vista de la partición que realizan del dominio de entrada y del dominio de salida.

Además de los dominios de entrada y de salida de cada componente, también se incorpora una salida de cada uno de ellos, que indica su tasa de fallos dado el perfil de funcionamiento en el que se hallan inscritos.

Por último, es necesario (dado el criterio de corrección topológica de Itacio) un componente que suministre los dominios de entrada al componente A, y también habrá sendos componentes de verificación cuya única misión es recibir el valor de la tasa de fallos y establecer los requisitos deseados.

De este modo, se obtiene la representación gráfica de la

Ilustración 38. Se aprecian los componentes A y B, el componente que representa el perfil operacional en el que se va a utilizar A, los componentes de verificación (VerifA y VerifB) que comprueban que la tasa de fallos de cada componente no supere la deseada, y los diferentes sumideros y fuentes (excepto las de tasa de fallos) representan la información sobre los subdominios de entrada y salida (no las entradas y salidas individuales).

Las garantías de InputForA establecen la información sobre el perfil operacional de A, que antes se ha denotado como $Q_A$. Las garantías de A, a su vez, realizan los cálculos de transformación de perfiles recogida en la Ilustración 37, así como el cálculo de la tasa de fallos del componente a partir del perfil operacional de entrada. B realiza operaciones análogas; en este caso la transformación del perfil operacional no se ha descrito en el ejemplo, pero sí se realizan los cálculos de tasa de fallos. Todo este funcionamiento se encuentra modelado mediante las oportunas expresiones restrictivas escritas en CLP.

En este caso, el sistema es correcto. La tasa máxima de fallos admisible se representa en este caso como un parámetro de los componentes del tipo FailVerif; en este caso, tanto VerifA como VerifB tienen la tasa máxima de fallos fijada en 0.02. Pero si se modifica el parámetro `maxFailRate` de VerifB para plantear mayores exigencias, disminuyendo dicha tasa máxima admisible hasta 0.01, el sistema detectará que, bajo el perfil operacional de A y dada la transformación de perfil que A realiza, el perfil operacional en el que se encuentra B no permite garantizar una tasa de fallos por debajo de 0.01. Como de costumbre, el error se señala y el usuario puede obtener una explicación sobre el problema, que consiste en que la tasa de error de B es en realidad 0.015132.

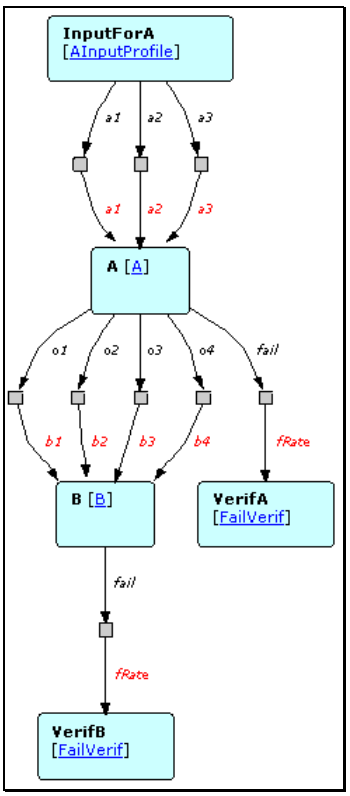



*Ilustración 38. Modelo de fiabilidad del ejemplo (captura de pantalla del prototipo Itacio-XDB).*

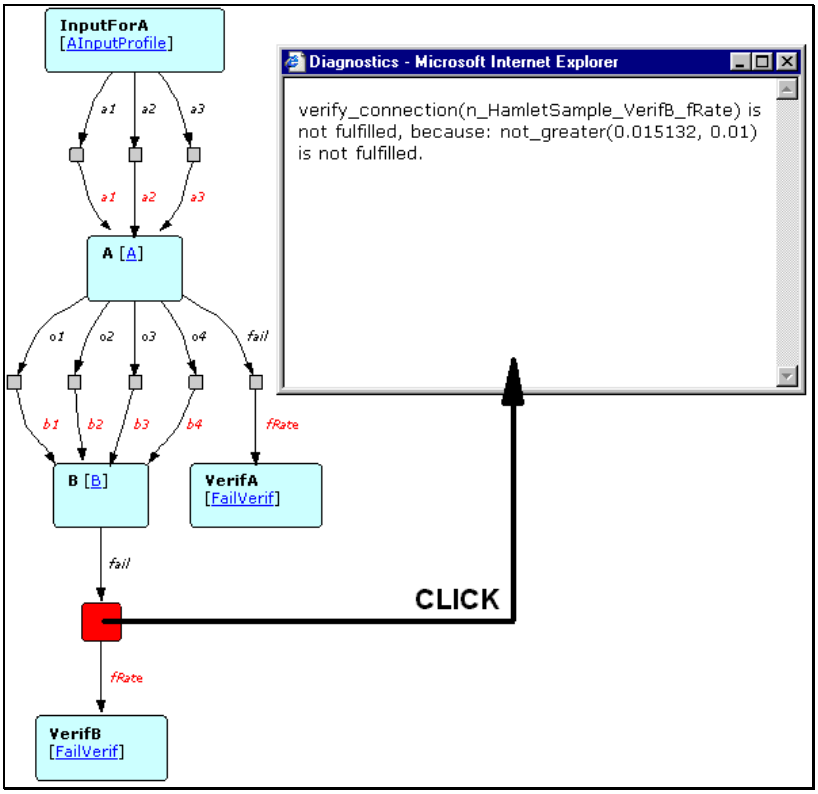



*Ilustración 39. Modelo de fiabilidad del ejemplo, con restricciones de fiabilidad más severas (capturas de pantalla del prototipo Itacio-XDB).*

### Más allá de la composición básica

Las operaciones descritas hasta ahora se corresponden con la llamada **composición básica** del modelo de Hamlet et al. Esta composición resuelve el problema fundamental, que es ofrecer en primer lugar un marco conceptual para modelar la fiabilidad de un componente y en segundo lugar un esquema para la composición de sucesivos componentes. Sin embargo, hay problemas adicionales a tener en cuenta, sobre todo cuando entra en juego la "lógica de interconexión" (*glue logic*).

Por ejemplo, se presenta el problema de la composición condicional de componentes. Considérese una composición como la siguiente:

$$B; \textbf{ if } b \textbf{ then } C_T \textbf{ else } C_F$$

¿Cómo encaja el modelo de Hamlet et al esta construcción? En realidad, la condición $b$ que se examina establece una partición del dominio; si al llegar al condicional existía un dominio D como entrada al siguiente componente, el condicional separa D en $D_T$ y $D_F$, incluyendo $D_T$ a todos los valores de D para los cuales la condición b se cumple y $D_F$ los valores para los que no se cumple. Basándose en esta idea, el condicional se puede evaluar reduciéndolo a la composición básica; se examinará la composición $B;C_T$ y tomando $D_T$ como subdominio de entrada a $C_T$, y se procederá análogamente con $C_F$. Esto permite calcular la

fiabilidad de $C_T$ y $C_F$. El perfil de salida del condicional puede calcularse estableciendo una media ponderada sobre los perfiles de salida de $C_T$ y $C_F$, donde la ponderación se obtiene a partir del dominio D y de la condición *b*.

Otro problema es el de los bucles. Los bucles de tipo while son examinados por Hamlet et al a la luz de las ideas presentadas para el condicional; la condición del bucle establece una partición en el dominio que percibirán los componentes internos al bucle. El dominio será a su vez transformado por los componentes internos al bucle, y el dominio transformado volverá a ser la entrada de la condición. Como es evidente, este análisis dista de ser trivial y los bucles son construcciones problemáticas en el análisis de programas; la opinión de Hamlet et al es que en esta aplicación concreta el problema no es tan grave, pero tampoco ofrecen una solución definitiva.

**Conclusiones**

Entre los muchos aspectos de los componentes software que puede ser oportuno especificar y verificar se encuentran las restricciones de fiabilidad. No obstante, en la actualidad no hay una tradición de ofrecer hojas de datos que describan la fiabilidad de los componentes. El modelo de Hamlet, Mason y Woit ofrece un marco básico (una teoría *fundacional*) en el que se inscriben las especificaciones y verificaciones de fiabilidad de los componentes software. En este caso, Hamlet et al identifican los componentes con subrutinas típicas por proporcionar una más fácil comprensión del mismo, aunque su intención es que este modelo sea aplicable a otros niveles en desarrollos futuros.

Haciéndose eco del interés que reviste en ocasiones el describir la fiabilidad de los componentes y su interacción, cabe esperar que un modelo como Itacio se aplique para este fin. El ejemplo anterior muestra cómo nuevamente es posible aplicar el modelo de forma casi directa a un nuevo y diferente propósito, en el que Itacio proporciona al modelo de Hamlet et al un soporte real.

Se trata de un caso más de aplicación de Itacio a fines que no estaban previstos cuando se concibió, dando cuenta de la flexibilidad y adaptabilidad del mismo para muy diversos propósitos de modelado de componentes. La operatividad en la práctica de esta teoría resulta cuestionable por diversos motivos. Las mediciones empíricas de la fiabilidad de los componentes pueden ser difíciles de llevar a cabo. Los cálculos de subdominios y, más aún, las operaciones que sobre ellos es necesario realizar para modelar construcciones condicionales y bucles, parecen particularmente complejos. El análisis de bucles es un problema abierto, y la teoría de Hamlet et al aún no ha sido llevada a su término. Pero ni la incompletitud de la teoría ni las dificultades prácticas para su aplicación son en realidad un problema de Itacio; el objeto de este experimento era sólo utilizar Itacio para representar un punto de vista más sobre los componentes (en este caso una concepción del estudio de la fiabilidad) cosa que ha resultado ser posible con relativa facilidad.

### 5.2.6. Verificación de compatibilidad entre protocolos según el modelo de Yellin y Strom

En el terreno de la interoperabilidad, como se ha repetido en esta tesis, existen diversos planos de verificación; en [CFPTV01] se considera adecuada una división típica entre verificaciones de signatura, verificaciones de protocolo (ordenación de las llamadas) y verificación funcional. La verificación de signaturas está muy presente en los entornos de

desarrollo habituales en la industria, y esta tesis se centra precisamente en abrir el proceso de verificación a otros objetivos. Los apartados precedentes ya ofrecen casos de verificación funcional de diversa índole. Sería interesante, pues, plantearse hasta qué punto Itacio es aplicable también, en el tercer frente, a casos de verificación de protocolos. Desde un punto de vista práctico, además, una fuente frecuente de errores es precisamente la realización de llamadas a un componente en un orden imprevisto, que conduce a dicho componente a un estado de error. Un caso típico es el olvidar realizar ciertas llamadas de inicialización o creación, o bien seguir utilizando el componente cuando este ya ha alcanzado un estado final; pero hay muchas más combinaciones reconocibles.

Existen diversos enfoques para el modelado formal de protocolos y para la verificación de dichos modelos; en el capítulo 2 de esta tesis se presentan algunas técnicas relacionadas de manera específica con estos fines. En el apartado actual, también como estudio práctico de la aplicabilidad de Itacio a este problema, se aborda el estudio de un modelo propuesto por Yellin y Strom [YS97]. La ventaja principal de este modelo de cara a nuestros fines es, precisamente, una mayor simplicidad que otros modelos formales rigurosos, lo que nos coloca en una situación más realista por lo que se refiere a las restricciones que en esta tesis nos hemos impuesto. En efecto, en Itacio se intenta en todo momento no obligar a los desarrolladores a abordar una formación profunda en técnicas formales específicas, sino aplicar conceptos sencillos de manera flexible y productiva; en este sentido, el modelo y la notación de Yellin y Strom resultan, probablemente, más asequibles que otros modelos formales, que serían más adecuados en ciertos ámbitos pero no tanto para el desarrollador medio. Hay que hacer notar que esto no necesariamente impide la aplicación de Itacio a estos otros modelos; por ejemplo, Yellin y Strom sostienen [YS97, pág. 300] que la mayor parte de su modelo se puede acomodar a la semántica de CSP.

**El modelo de Yellin y Strom**

El planteamiento inicial de Yellin y Strom es muy similar al de esta tesis. El uso de componentes software va extendiéndose progresivamente, pero está pendiente de resolver el problema de especificar las interfaces de un componente de modo que se pueda saber si es un *compañero válido* de otro (si funcionarán adecuadamente una vez conectados). Estos autores abordan un problema adicional: cómo crear *adaptadores* cuando los dos componentes son *funcionalmente compatibles* pero sus respectivas interfaces visibles no lo son. En este documento nos centraremos exclusivamente en el primer problema, que es el que guarda relación directa con nuestros propios objetivos.

En su análisis preliminar de las tecnologías disponibles, Yellin y Strom llegan a la misma conclusión de partida que esta tesis: la especificación de interfaces en un entorno como, por ejemplo, CORBA, sólo permite definir métodos disponibles en cada componente, pero no restricciones adicionales. En lo referente al segundo nivel de verificación mencionado anteriormente, IDL y similares no permiten reflejar restricciones sobre el orden en que deben enviarse ciertos mensajes.

Los autores, a raíz de esto, formulan una noción de compatibilidad entre protocolos, que permite detectar problemas que el sistema de tipos y signaturas por sí mismo no puede detectar (como el interbloqueo, por ejemplo). Se define una forma de describir los protocolos, que puede añadirse al sistema de tipos tradicional (es decir, se complementa la verificación de signaturas con la verificación de protocolos). El modelo que proponen es parecido al propuesto anteriormente en trabajos de Nierstrasz o de Allen y Garlan, pero

con ciertas diferencias importantes. En primer lugar, la especificación de protocolos de Yellin y Strom describe tanto los mensajes emitidos como los recibidos por cada componente. En segundo lugar, la semántica de componibilidad de Yellin y Strom es bastante simple, lo que tiene como consecuencia negativa que se restringe un tanto el número de protocolos que resultan compatibles, pero a cambio se obtiene una verificación de protocolos muy sencilla, que no requiere tecnologías complejas y puede integrarse fácilmente con el sistema de tipos preexistente. Y en tercer lugar, no se requiere un lenguaje o entorno específico para implementar el sistema; lo que se proporciona es una estrategia de implementación que permite fusionar la semántica de protocolos con muy diversos lenguajes y entornos ya existentes. Está claro que este tercer punto es particularmente importante para los objetivos planteados en Itacio, además de que se inscribe en la misma línea de actuación que este.

Al especificar protocolos, es frecuente adoptar una semántica asíncrona. No obstante, la semántica asíncrona plantea dificultades adicionales, y es más fácil analizar los sistemas bajo una semántica síncrona.

**Especificación de protocolos**

Se supone que cada componente expone interfaces dotadas de especificación de tipos, y a través de estas interfaces envía y recibe mensajes que intercambia con un potencial componente colaborador (*compañero* de colaboración). Cada componente puede exponer múltiples interfaces, lo que le permite colaborar a la vez con diversos componentes colaboradores; no obstante, es posible modelar estas colaboraciones múltiples como una suma de colaboraciones bilaterales. La especificación de una colaboración, en este modelo, consta de dos partes:

- La *signatura de la intefaz* describe el conjunto de mensajes que un componente puede intercambiar con su compañero. Para cada mensaje se indican los parámetros asociados, y también si es un mensaje enviado o recibido.
- El *protocolo* describe un conjunto de *restricciones de secuencia*. Estas restricciones definen cuáles son las formas legales de ordenar los mensajes, mediante una gramática de estados finitos. Esta gramática consta de un conjunto de estados y un conjunto de transiciones; cada transición se asocia con un mensaje enviado o recibido en un estado concreto.

Una transición tiene la forma general:

```
<estado> : <dirección> <mensaje> -> <estado>
```

donde `<estado>` es el nombre simbólico de un estado, `<dirección>` (denominada a veces *polaridad*) es "+" si el mensaje se recibe y "-" si el mensaje se envía, y `<mensaje>` es el nombre de un mensaje que esté descrito en la signatura. Todos los protocolos tienen un estado inicial único, $init_p$. Las transiciones son deterministas en el sentido de que no se permite que de un estado dado partan dos transiciones que lleven a estados distintos pero tengan asociado el envío del mismo mensaje (o bien la recepción del mismo mensaje). Las únicas transiciones permitidas y los únicos mensajes que se pueden enviar y recibir en cada estado están, pues, recogidos en la especificación del protocolo.

Por lo que respecta a estados finales, el protocolo también puede tener una serie de estados considerados finales (de los cuales no parte ninguna nueva transición) o bien no tener estados finales, con lo que funcionaría indefinidamente.

Para evitar situaciones de falta de funcionamiento arbitrarias, se da por supuesto que un componente no permanecerá indefinidamente en un estado en el que pueda realizar una transición de envío de mensaje. De este modo se evita la hipotética situación de que un componente esté en un estado en el que pueda enviar cierto mensaje, su compañero esté en un estado en el que pueda recibir ese mismo mensaje, y sin embargo el sistema no progrese de forma alguna.

Yellin y Strom definen, partiendo de estos conceptos elementales, una notación para describir el protocolo de un componente. Puede verse un ejemplo (el protocolo de un filtro) en la Ilustración 40.

```
Collaboration Filter {
   Receive Messages {
      itemToBeFiltered(dataItem:ObjectRef);
      noMoreItems();
   };
   Send Messages {
      newFilterRequest();
      ok();
      remove();
   };
   Protocol {
      States {Stable(init),Filter,Respond};
      Transitions {
         Stable: -newFilterRequest -> Filter;
         Filter: +itemToBeFiltered -> Respond;
         Filter: +noMoreItems -> Stable;
         Respond -ok -> Filter;
         Respond: -remove -> Filter;
      };
   };
};
```

*Ilustración 40. Ejemplo de especificación de protocolo (Yellin & Strom).*

**Semántica síncrona**

Ya se ha comentado que los protocolos de este modelo se adhieren a una semántica síncrona. En la semántica asíncrona, si un componente se encuentra en un estado con una transición asociada al envío de un mensaje, el componente puede enviar ese mensaje en cualquier momento, independientemente del estado del componente colaborador. Esto obliga a que cada componente tenga asociada una cola en la que almacena los mensajes que le han sido enviados, pero aún no ha procesado. No es difícil implementar sistemas asíncronos, pero sí es difícil razonar sobre ellos; algunas importantes propiedades de estos sistemas son, en términos generales, indecidibles (como es el caso de los interbloqueos).

En la semántica síncrona, sin embargo, un componente sólo puede enviar un mensaje si el receptor se encuentra en un estado en el que pueda procesar dicho mensaje. Las máquinas de estados de ambos componentes avanzan de manera sincronizada (no son necesarias

colas como en el caso anterior). Resulta mucho más fácil razonar sobre sistemas síncronos. De hecho, no es imprescindible que el envío y recepción de un mensaje formen realmente una operación "atómica", sino que basta con que los dos componentes respeten una *traza de ejecución* común (el orden de los mensajes enviados y recibidos).

**Compatibilidad de protocolos**

Yellin y Strom definen primero un criterio de compatibilidad de protocolos, y ofrecen un sencillo algoritmo para la verificación de dicha compatibilidad.

Básicamente, los problemas que afectan a la compatibilidad de protocolos son dos, y se toma esta definición de trabajos anteriores de Brand y Zafiropulo [BZ83]:

- **Recepciones no especificadas** (*unspecified receptions*): Se produce una recepción no especificada cuando un componente está en un estado en el que no puede recibir cierto mensaje, mientras que su compañero alcanza un estado en el que envía dicho mensaje. Dicho de otro modo, un componente recibe un mensaje que no está preparado para recibir.
- **Interbloqueos** (*deadlocks*): Se produce un interbloqueo cuando los protocolos de los dos componentes no alcanzan un estado final, es decir, si ambos llegan a sendos estados que no son finales y en los cuales la colaboración no puede continuar.

Examinemos ahora la notación. Los estados de un protocolo $P$ se denotan por *States(P)*, y las transiciones por *Transitions(P)*. Dado un mensaje $m$, la función auxiliar *Polarity(P, m)* devuelve + si $m$ es un mensaje recibido en el protocolo $P$, y - si es un mensaje enviado. Un *estado de colaboración* para los protocolos $P_1$ y $P_2$ es un par $\langle s, t \rangle$ tal que $s \in States(P_1)$ y $t \in States(P_2)$. Una *historia de colaboración* para los protocolos $P_1$ y $P_2$ es una secuencia (posiblemente infinita) de la forma $\alpha_1 \rightarrow m_1\ \alpha_2 \rightarrow m_2 \cdots$ donde:

- Cada $\alpha_i$ es un estado de colaboración para $P_1$ y $P_2$.
- $\alpha_1 = \langle init_{P1}, init_{P2} \rangle$, y
- $\alpha_{i+1} = \langle s_{i+1}, t_{i+1} \rangle \Leftrightarrow \alpha_i = \langle s_i, t_i \rangle$ y $(s_i : m_i \rightarrow s_{i+1}) \in Transitions(P_1)$, $(t_i : m_i \rightarrow t_{i+1}) \in Transitions(P_2)$, y $Polarity(P_1, m_i) \neq Polarity(P_2, m_i)$.

Por definición, $Collabs(P_1, P_2) = \{\alpha : \alpha$ es una historia de colaboración para $P_1$ y $P_2\}$. Dicho de otro modo, $Collabs(P_1, P_2)$ contiene todas las trazas que pueden darse cuando $P_1$ y $P_2$ colaboran.

Yellin y Strom ofrecen una verificación de la existencia de interbloqueos y de recepciones no especificadas en términos de elementos del conjunto $Collabs(P_1, P_2)$, extraída de [BZ83]. A su vez, **dos protocolos son compatibles si no tienen recepciones no especificadas ni interbloqueos**.

Dados $s \in States(P_1)$, $t \in States(P_2)$, por definición $s \sim t$ sii existe una historia de colaboración de la cual forme parte $\langle s_i, t_i \rangle$. De este modo, la relación $\sim$ recoge los pares de estados que pueden darse en una colaboración entre $P_1$ y $P_2$. Esto abre las puertas a verificar la compatibilidad entre protocolos simplemente mediante el cálculo de todos los posibles pares $s \sim t$, y aplicando a dichos pares los criterios de recepción no especificada e interbloqueo. Para calcular los pares $s \sim t$, Yellin y Strom definen en [YS97] un algoritmo

para el cálculo de cierto conjunto $EQUIV(P_1, P_2)$, y prueban después que este conjunto es equivalente a la relación ~.

**Aplicación de Itacio**

Dadas estas definiciones y algoritmos, el siguiente paso es la instanciación de Itacio para el problema concreto de la verificación de protocolos. Un componente según Yellin y Strom puede exhibir varias interfaces, y cada interfaz se asocia con un protocolo que describe la ordenación legal de mensajes que la afectan.

En relación con Itacio, podría entenderse que cada componente exhibe como fuentes los mensajes enviados, y como sumideros los mensajes recibidos. Este enfoque encajaría con una forma habitual de modelar los componentes software (de manera similar a los objetos) y permitiría además establecer expresiones restrictivas sobre esas fuentes y sumideros de manera individual, dando pie a realizar las verificaciones habituales de signaturas, o bien otras verificaciones funcionales más variadas, como las presentadas en otros ejemplos de esta disertación.

Además de esa estructura a nivel de mensaje, es necesario ofrecer información de tipo global a la interfaz, puesto que además de los mensajes por separado se ha de verificar también el protocolo como un todo. En este caso, en aras de la simplicidad, representaremos los componentes sólo con esta información de tipo global; cada componente tendrá una fuente que representa toda la información sobre el protocolo, y obviaremos las fuentes y sumideros de los mensajes (aunque nada impediría representarlos también). Se trata, pues, de un enfoque parecido al presentado para los contratos de reutilización (pág. 130), en el que se realizan verificaciones sobre un tipo de datos relativamente complejo, que en aquel caso eran contratos de reutilización y en este protocolos. Al igual que entonces, se introduce un componente que ejerce de mediador entre los dos componentes compañeros de colaboración, y que realiza la verificación de compatibilidad entre protocolos.

Realizada la instanciación del modelo, sólo resta representar el conocimiento disponible mediante las expresiones restrictivas, e incluir en la biblioteca del sistema la representación (en programación declarativa) del algoritmo que verifica la compatibilidad entre protocolos.

**Ejemplo de representación de protocolos**

Como ejemplo, se ha elegido un sistema bastante habitual en el que un componente ejerce de "fichero", de modo que proporciona una corriente de caracteres a un "lector". La abstracción de un fichero es un caso típico de ordenación de eventos: hay que abrir un fichero antes de leer caracteres del mismo. Si se combina esto con la programación orientada a objetos, también cobra importancia la invocación del constructor del fichero antes de abrirlo, y olvidar alguno de estos pasos es un error de programación muy común, sobre todo entre principiantes.

Como primer paso, se ha modelado el protocolo de un fichero (file) y el de un lector de ficheros (fileReader). El protocolo de file puede verse en la Ilustración 41; el estado inicial es initFile, y cada transición lleva como etiqueta el mensaje enviado (polaridad -) o recibido (polaridad +).

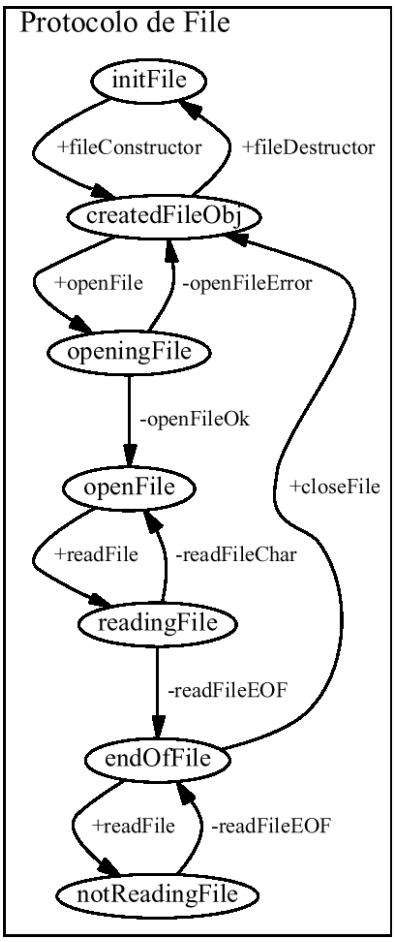



*Ilustración 41. Protocolo del objeto file.*

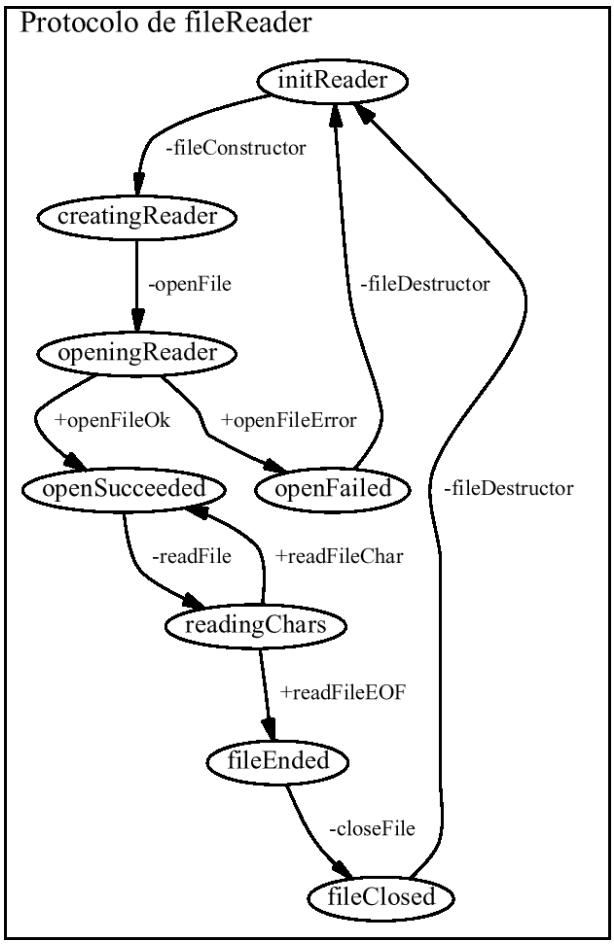



*Ilustración 42. Protocolo del objeto fileReader.*

El protocolo del componente fileReader se ha representado en la Ilustración 42, siguiendo las mismas convenciones gráficas. Puede verse que en este caso se trata de protocolos que no terminan nunca, es decir, carecen de estados finales; se supone que una vez cerrado y destruido un objeto file, puede volver a crearse una instancia. No habría ningún problema en modelar estos protocolos con estados finales.

Esta es, pues, la representación de ambos protocolos en forma gráfica. Sólo restaría expresar este conocimiento en forma de expresiones restrictivas en Itacio. Un ejemplo de esto es el protocolo del objeto file, que se representa en la Ilustración 43; puede verse que la naturaleza declarativa de la especificación de protocolos de Yellin y Strom encuentra fácil acomodo en la sintaxis de las cláusulas Horn. El segundo predicado de la figura detalla el estado inicial (único), la lista de estados finales (en este caso vacía) y la lista de los demás estados. A continuación figuran los mensajes enviados y recibidos, y las transiciones, que incluyen el estado de partida, la polaridad (+ ó -), el mensaje y el estado de llegada.

```
ys_collaboration($file$).
ys_states($file$, initFile, [], [createdFileObj, openingFile,
      openFile,readingFile, endOfFile, notReadingFile]).
ys_sent_message($file$, openFileError).
ys_sent_message($file$, openFileOk).
ys_sent_message($file$, readFileChar).
ys_sent_message($file$, readFileEOF).
ys_received_message($file$, fileConstructor).
ys_received_message($file$, fileDestructor).
ys_received_message($file$, openFile).
ys_received_message($file$, readFile).
ys_received_message($file$, closeFile).
ys_transition($file$, initFile, +, fileConstructor,
      createdFileObj).
ys_transition($file$, createdFileObj, +, fileDestructor, initFile).
ys_transition($file$, createdFileObj, +, openFile, openingFile).
ys_transition($file$, openingFile, -, openFileError,
      createdFileObj).
ys_transition($file$, openingFile, -, openFileOk, openFile).
ys_transition($file$, openFile, +, readFile, readingFile).
ys_transition($file$, readingFile, -, readFileChar, openFile).
ys_transition($file$, readingFile, -, readFileEOF, endOfFile).
ys_transition($file$, endOfFile, +, readFile, notReadingFile).
ys_transition($file$, notReadingFile, -, readFileEOF, endOfFile).
ys_transition($file$, endOfFile, +, closeFile, createdFileObj).
```

*Ilustración 43. Descripción mediante cláusulas Horn del protocolo de la Ilustración 41.*

De manera análoga se puede representar con facilidad el protocolo de fileReader. Basta introducir en Itacio la descripción de ambos componentes, así como del componente de unión que verifica la compatibilidad entre protocolos, y se obtiene un gráfico como el de la Ilustración 44.

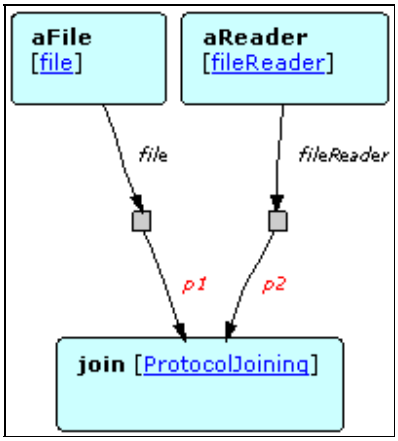



*Ilustración 44. Un sistema de protocolos correcto (captura de pantalla del prototipo Itacio-XDB).*

En este caso, ambos protocolos son compatibles. Pero supóngase que se ha cometido el error de no abrir el fichero, y de diseñar un protocolo de lector que se limita a crear el objeto fichero y comenzar a leer caracteres. Dicho protocolo podría tener la estructura de la Ilustración 45. La representación de este protocolo en Itacio sería totalmente análoga a lo presentado para los protocolos correctos.

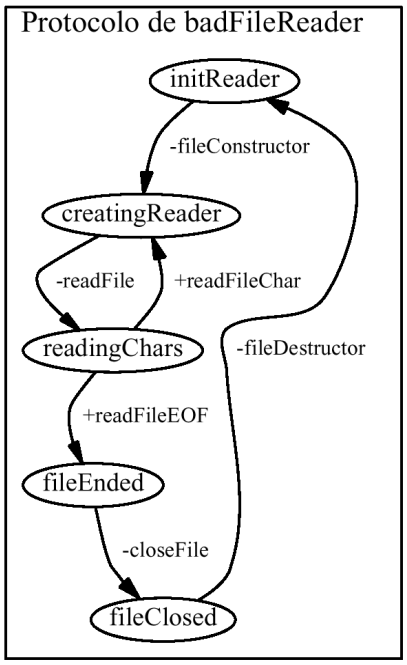



*Ilustración 45. Protocolo de badFileReader.*

Una vez definido el componente, se puede introducir una instancia suya en el sistema existente. Si esta instancia se conecta con la del protocolo de fichero y se introduce una comprobación, el sistema quedaría como en la Ilustración 46. En efecto, si se interroga a la base de conocimientos, se comprueba que entre los protocolos badFileReader y file existe una incompatibilidad doble: el par de estados createdFileObj de file y creatingReader de badFileReader adolecen de interbloqueo y de recepción no especificada (Ilustración 47) Véase la Ilustración 41, y se comprobará que en efecto el hecho de no abrir el fichero implica que el protocolo de file nunca avanza del estado createdFileObj al estado siguiente (openingFile) y que además el mensaje que file recibe en el estado createdFileObj por parte de badFileReader es readFile, que no está preparado para recibir.

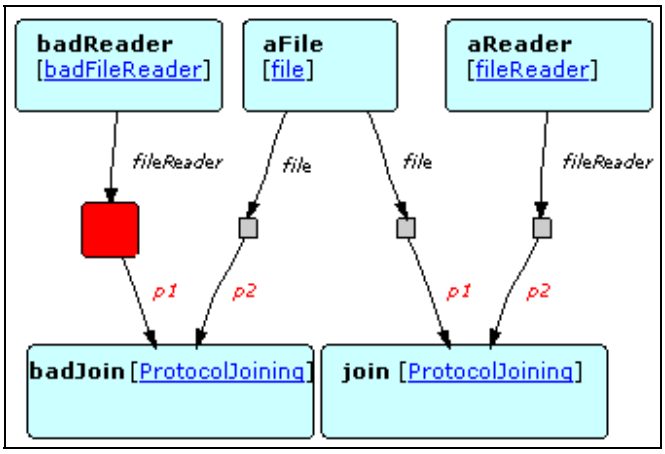



*Ilustración 46. Ejemplo de protocolos incompatibles en Itacio-XDB.*

```
ECLiPSe Constraint Logic Programming System [kernel]
Copyright Imperial College London and ICL
Certain libraries copyright Parc Technologies Ltd
GMP library copyright Free Software Foundation
Version 5.1.0, Wed Nov 22 12:31 2000
[eclipse 1]: ys_protocols_compatible(file, badFileReader).

no (more) solution.
[eclipse 2]: ys_deadlock_pairs(file, badFileReader, X).

X = [[createdFileObj, creatingReader]]
yes.
[eclipse 3]: ys_unspecified_reception_pairs(file, badFileReader, X).

X = [[createdFileObj, creatingReader]]
yes.
```

*Ilustración 47. Una traza de la interrogación de la base de conocimientos en ECLiPSe.*

El algoritmo de cálculo para detectar los interbloqueos y las recepciones no especificadas se encuentra en una biblioteca al efecto, y su implementación en CLP es casi trivial (se trata de una mera traslación de lo presentado en [YS97] a la programación lógica).

### Conclusiones

Nuevamente, se comprueba que la traslación de Itacio a diferentes conceptos y modelos relacionados con los componentes resulta relativamente sencilla. Habiéndose cubierto ya previamente los aspectos de verificación clásica de signaturas y otros aspectos de compatibilidad funcional, resultaba interesante comprobar si podía realizarse la verificación de las restricciones que afectan al orden de las invocaciones, un problema típico en el ensamblaje de componentes.

Básicamente, se trata de verificar la compatibilidad de protocolos. Este es un campo muy amplio, y existen diversas técnicas de descripción formal y diversos modelos que ya se han presentado en capítulos anteriores de esta disertación, evaluando sus puntos fuertes y su grado de adecuación a los objetivos de esta tesis. No todos estos métodos encontrarán un reflejo trivial en Itacio, y pueden requerir mayor o menor esfuerzo de instanciación o incluso no ser trasladables en su totalidad. Pero lo que sí se demuestra es que lo que se puede expresar mediante lógica de primer orden no resulta particularmente difícil de llevar a Itacio.

En este caso, se ha elegido un modelo de verificación de protocolos bastante operativo y comprensible, en el que Yellin y Strom ofrecen ya una solución algorítmica. Partiendo de esta base, se ha podido comprobar en la práctica que la verificación de compatibilidad de protocolos mediante Itacio no plantea dificultades notables.

## 5.3. Escrutinio público

A lo largo de esta tesis se han utilizado diferentes medios para ponderar tanto la corrección como la originalidad. Uno es el estudio comparativo y el análisis del estado del arte en los campos relacionados (capítulos 2 y 3.2). Otro, la realización de estudios experimentales a través de diversos prototipos (capítulos 5.1 y 5.2). Otra vía, la presentada aquí, es la publicación de diversos aspectos de este trabajo y su presentación en diversos foros para someterlo a la crítica y análisis de otros investigadores y profesionales expertos en áreas relacionadas.

En este capítulo se describirán estas actividades de divulgación y presentación, que abarcan diferentes aspectos de Itacio: componentes software, gestión del conocimiento, ingeniería del software, programación lógica, etc. El hecho de que Itacio integre estas diferentes disciplinas hacía que fuese adecuado recabar la opinión de expertos desde estos diferentes puntos de vista, y así se ha intentado hacer en los diferentes foros a los que se ha acudido.

### 5.3.1. III Workshop on Component-Based Software Engineering – 22nd International Conference on Software Engineering

El 5 y 6 de junio de 2000 tuvo lugar en la Universidad de Limerick (Irlanda) la vigésima segunda conferencia internacional sobre ingeniería del software (ICSE 2000), un evento de gran prestigio. Dentro del programa de actividades de este ciclo de conferencias, se celebró la tercera edición del taller sobre ingeniería del software basado en componentes (CBSE 2000).

Dicho taller, dirigido por Kurt Wallnau, del Software Engineering Institute de la Universidad Carnegie Mellon, pretendía centrarse en el estudio de sistemas de componentes "del mundo real", y reunía a investigadores y profesionales en el campo de la tecnología de componentes. Se trataba de un taller cerrado, en el que los participantes realizaban una rápida exposición y el resto del tiempo estaba dedicado a discusiones, preguntas y debates sobre las ponencias. El carácter de este taller, tanto por su temática como por su formato como por el perfil de los participantes, lo convertía en un foro especialmente adecuado para la presentación de Itacio, ya que allí estarían reunidos expertos en componentes software a nivel mundial y representantes muy significativos de la industria del desarrollo de software basado en componentes.

El artículo presentado [CLC00] fue una visión general del modelo, que recogía prácticamente la primera organización coherente de las ideas que habían dado lugar a la tesis, y que era el resultado de las experiencias con el prototipo Itacio-SEDA (capítulo5.1.1). El propósito de esta participación era básicamente tener una referencia de la opinión de otros investigadores y profesionales especialistas en componentes software.

De unos 40 artículos presentados, sólo 17 fueron aceptados para su defensa en el taller, lo que fue una primera señal de que las ideas contenidas en el manuscrito eran significativas y originales. La presentación en CBSE 2000 resultó extremadamente positiva, ya que a pesar de que el formato favorecía la crítica y la discusión de las ideas planteadas (y así ocurrió en el caso de algunas ponencias, cuyos puntos flacos fueron señalados con insistencia) el modelo Itacio fue acogido con general aceptación por parte de investigadores de perfil muy

diverso y contrastada experiencia. Esto confirmó que las suposiciones en las que se basaba la tesis eran básicamente correctas, o al menos que no presentaban deficiencias obvias, y que se estaba trabajando en la dirección adecuada; fue el mejor espaldarazo que esta tesis podía recibir en lo que respecta a revisión independiente.

### 5.3.2. Software Composition Group – Universidad de Berna

En marzo de 2001 el Software Composition Group de la Universidad de Berna [SCG] recabó información sobre el modelo Itacio, a raíz de la ponencia en el CBSE / ICSE 2000. Asimismo, el grupo PECOS dentro del SCG solicitó información más detallada sobre el modelo de verificación.

### 5.3.3. Workshop on Knowledge Management – 4th International Symposium, Soft Computing / Intelligent Systems for Industry

Evidentemente, el modelo Itacio incorpora ideas relacionadas con la gestión del conocimiento. Se pretende que este método permita a los desarrolladores recoger el conocimiento que se tiene sobre los componentes, y utilizarlo de manera efectiva en un proceso de verificación automatizado.

El simposio SOCO / ISFI se centra en aspectos de computación flexible (redes neuronales, algoritmos genéticos, etc.) y aplicación de sistemas informáticos "inteligentes" a la industria. Dentro del congreso SOCO / ISFI 2001, celebrado en la Universidad de Paisley (Escocia) del 26 al 29 de junio de 2001, tuvo lugar un taller (de un día de duración) sobre Gestión del Conocimiento. Se decidió que era una buena ocasión para someter Itacio a la consideración de investigadores que trabajasen con esta orientación.

La ponencia fue aceptada [CLC01a] y se presentó una descripción de Itacio centrada en los aspectos de gestión del conocimiento, lo que comprende la generación de la base de conocimientos y el proceso de inferencia. Este artículo recogía un modelo más refinado que el presentado en ICSE 2000, fruto de los experimentos con el nuevo prototipo Itacio-XJ. Nuevamente, las opiniones recogidas fueron positivas y durante la interacción con otros investigadores estos no identificaron puntos débiles -fuera de los ya previstos y conocidos como fruto de nuestro propio trabajo- ni falta de originalidad en lo presentado.

### 5.3.4. Simposio Iberoamericano de Sistemas de Información e Ingeniería del Software en la Sociedad del Conocimiento

Entre el 29 y el 31 de agosto de 2001 tuvo lugar en la Universidad Distrital "Francisco José de Caldas" de Bogotá (Colombia) el simposio SISOFT 2001. Este foro, cuyo enfoque englobaba la ingeniería del software y la gestión del conocimiento, también parecía apropiado para someter a Itacio a la opinión de otros expertos. La ponencia presentada [CLC01b] constituía básicamente una revisión general del modelo, adaptada a las áreas de interés del simposio, con el fin de exponerlo nuevamente a la revisión de otros investigadores, en este caso de ámbito iberoamericano. Nuevamente, la experiencia fue positiva, acogiéndose la ponencia sin opiniones contrarias al planteamiento de Itacio.

### 5.3.5. Jornadas de Transferencia de Tecnología de la Universidad de Oviedo

El Vicerrectorado de Investigación de la Universidad de Oviedo convocó a los diferentes grupos de investigación para que presentasen ideas y proyectos en la llamada Oferta Tecnológica. Esta oferta incluye la celebración de unas jornadas, presentación de pósteres y edición de un catálogo de oferta tecnológica, oferta que se dirige al sector empresarial y público. Los contenidos incluidos en la oferta son los que se espera que pueden tener una mayor acogida externa.

Aunque Itacio podía ofrecerse globalmente, se pensó que de cara a formar parte de una oferta concreta era más apropiado elegir algún subproyecto específico, más fácil de identificar por parte de personas potencialmente interesadas. De este modo, se presentó un póster [Ce01] que proponía emplear el modelo Itacio para el diagnóstico remoto de equipos sobre el sistema operativo Windows. Este particular se discute en el capítulo 5.2.3.

### 5.3.6. VI Jornadas de Ingeniería del Software y Bases de Datos

Las Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2001) ofrecían un marco apropiado para presentar algunos aspectos de aplicación de Itacio, especialmente aquellos relevantes bajo un enfoque centrado en la ingeniería del software. Puesto que Itacio ya había sido evaluado por personal especialista en componentes software y en gestión del conocimiento, el someter a discusión estas ideas ante una audiencia de un perfil más relacionado con la ingeniería del software podía aportar puntos de vista nuevos a la investigación.

Se presentó un artículo [CLC01c] sobre la aplicación de Itacio para modelar y verificar la evolución de los contratos de reutilización (tema descrito en el capítulo 5.2.2. de este documento). En el JISBD 2001 se aceptaron 38 artículos (incluido el aquí referenciado) de unos 90 presentados. Nuevamente, la revisión fue básicamente positiva. La presentación en las Jornadas suscitó el interés de los presentes y diversos comentarios favorables a la idea; tampoco se cuestionó su originalidad ni su viabilidad.

### 5.3.7. 17th International Conference on Logic Programming

Continuando con la línea de actuación adoptada inicialmente, que consiste en someter a escrutinio público el modelo Itacio ante comunidades especializadas en los diferentes aspectos del mismo, faltaba por presentar nuestro trabajo en un foro relacionado con la programación lógica, que aunque no es realmente la motivación básica del proyecto sí que es una tecnología claramente involucrada en el mismo.

Como parte de la decimoséptima Conferencia Internacional sobre Programación Lógica, el día 1 de diciembre de 2001 se celebraba un taller sobre programación lógica con restricciones y su relación con la ingeniería del software (CLPSE'01, la segunda edición del CLPSE). Evidentemente, resultaba de gran interés conocer la opinión de los organizadores y participantes en este taller, puesto que su temática y características encajaban a la perfección con el papel que la programación lógica con restricciones juega en esta tesis.

Se remitió un artículo [CLC01d] sobre la aplicación de Itacio a un sistema de componentes convencional, dedicado al procesamiento de sonido en tiempo real (este caso de aplicación se describe en este documento en el capítulo 5.2.4). Este artículo fue aceptado con

comentarios muy positivos; la presentación en el taller también suscitó el interés de los presentes y críticas favorables. Además, la presencia en este taller sirvió para contactar con otros investigadores con ideas similares sobre la aplicación de la programación lógica a la ingeniería del software, cosa que no había sucedido en los foros anteriores.

### 5.3.8. 5º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software – IDEAS 2002-03-21

En el momento de generar este documento, un artículo sobre la aplicación de Itacio al modelo de verificación de Hamlet et al (véase apartado 5.2.5) ha sido aceptado para su presentación en la quinta edición del Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS 2002) a celebrar en La Habana.

# 6. Conclusiones y trabajo futuro

## 6.1. Principales contribuciones

El desarrollo de esta tesis ha dado como resultado una serie de aportaciones en el campo de la verificación de sistemas de componentes que el autor entiende que son originales, basándose principalmente en:

1. El análisis del estado del arte realizado. No se ha encontrado una propuesta que cumpliese las mismas características.
2. El escrutinio público al que se ha sometido este proyecto. Presentado en foros que cubrían diversas ramificaciones del mismo (componentes, ingeniería del software, programación lógica, gestión del conocimiento) los expertos presentes no han señalado la existencia de una propuesta similar.

Las principales contribuciones de esta tesis pueden resumirse como sigue:

- Tecnología de componentes software
    - Un estudio de las alternativas existentes para la verificación de la composición de componentes software desde el punto de vista de los condicionantes planteados aquí (verificación estática, automática y asequible).
    - Una definición de componente abierta y aplicable en muy diversos ámbitos del diseño, desarrollo y mantenimiento del software.
- Gestión del conocimiento aplicada a componentes software
    - Un modelo de componente que permite incorporar al mismo el conocimiento disponible, yendo este mucho más allá de las meras signaturas e interfaces clásicas.
    - Un esquema de generación de la base de conocimientos que permite integrar el conocimiento sobre los diferentes componentes y realizar un proceso de inferencia que tenga en cuenta las relaciones concretas entre los mismos (ya sean directas o indirectas) en el sistema que se verifica.
- Ingeniería del software
    - Un método de modelado de componentes cuya finalidad es la verificación, y que puede ser aplicado sobre entidades de muy diferentes tipos y en

etapas diversas del ciclo de vida del software. Este método de modelado y verificación tiene además características especiales:

  - Puede ser utilizado con facilidad por una organización de desarrollo, sin necesidad de conocimientos sobre métodos formales.
  - Fomenta la recogida y uso de conocimiento que habitualmente se pierde.
  - Permite gestionar este conocimiento sin necesidad de integrarlo en el código fuente de los programas desarrollados.
  - Se puede utilizar sin necesidad de desarrollar realmente el programa objeto de estudio.
  - No está ligado a ningún lenguaje de desarrollo específico ni a ningún propósito específico.

- Ejemplos de aplicación de un esquema de componentes a diversos terrenos del desarrollo de software, como la verificación de contratos de reutilización o de restricciones de fiabilidad.
- Un sistema de verificación de componentes plenamente viable en la práctica:
  - Las herramientas pueden desarrollarse con base en tecnologías ampliamente disponibles y conocidas.
  - Los conocimientos básicos necesarios encajan en el perfil típico de los profesionales del desarrollo de software, no planteando un gran choque de mentalidad su adquisición en caso necesario.
- Diversos prototipos que apoyan la viabilidad de los métodos propuestos.

## 6.2. Conclusiones

El desarrollo de software basado en componentes suele recurrir a descripciones de las interfaces de los componentes, entendidas (con las variaciones procedentes en cada caso) como una colección de métodos y de parámetros que reciben (las denominadas *signaturas*). Cuando se combinan componentes, es frecuente que se verifique que dichas signaturas encajan correctamente, y existen medios (que han alcanzado plena difusión) para detectar esto en etapas tempranas de ese proceso de combinación.

Sin embargo, existen defectos que no tienen relación alguna con las signaturas de los componentes, y que surgen también de su combinación. Y estos defectos no suelen detectarse de ninguna forma automatizada; no parece haber alcanzado amplia difusión ningún método encaminado a tener en cuenta las restricciones de funcionamiento los componentes y a poner en relación esas respectivas restricciones, sean del tipo que sean (funcionales, de rendimiento, de propósito, de dominios, etc.) La experiencia profesional demuestra que en muchas ocasiones estos defectos se detectan en la fase de pruebas, o peor aún, en la fase de explotación.

Muchos de estos defectos podrían detectarse en etapas tan tempranas como las incongruencias de signaturas. Sólo se necesitaría un método que permitiese adjuntar a cada componente una descripción de las restricciones conocidas respecto a su funcionamiento, y utilizar esas restricciones en un sistema de verificación.

Este método debería presentar ciertas propiedades importantes. La verificación debería ser estática en el sentido descrito; no se trata de probar el sistema en ejecución, sino de llegar a conclusiones mediante el puro análisis a priori de la información disponible.

La verificación debe también ser automática porque eso es lo que posibilita que sea realizada repetidamente, a un bajo coste, y que ofrezca resultados no ambiguos.

Algunos métodos formales (u otras técnicas) podrían ser utilizados para propósitos similares a los planteados aquí; pero la percepción que se tiene de los métodos formales y su dificultad de adopción impiden en la práctica una difusión a gran escala. El método buscado debería observar en todo momento como prioritaria la simplicidad, viabilidad técnica, flexibilidad de aplicación y facilidad de adopción que otras técnicas no contemplan.

La tesis aquí planteada es que es posible dotar a la tecnología de componentes de esos métodos que a nuestro juicio necesita. Y la demostración de esta tesis se ha basado en la construcción de un modelo y un método que responde a los requisitos expuestos. La validación del método desarrollado se basa en el desarrollo de prototipos del sistema de verificación, la aplicación de esos prototipos a diferentes tipos de problemas, y el sometimiento del modelo al dictamen de expertos en los diferentes ámbitos involucrados.

El proceso de desarrollo de prototipos, siempre mediante técnicas fiables y ampliamente disponibles, ha servido para constatar que el desarrollo de herramientas que soporten el método es técnicamente viable.

La aplicación de esos prototipos a problemas muy diferentes ha permitido comprobar la flexibilidad y amplitud de miras del método. Deliberadamente, se han elegido casos de estudio variados, no relacionados entre sí, representativos de diversos aspectos del desarrollo del software, y sobre todo se trata de casos de estudio que en absoluto se habían tenido en cuenta a la hora de formular el modelo o las herramientas. La aplicación de Itacio a estos casos de estudio diversos e imprevistos ha sido posible con enorme facilidad, lo que constituye una clara muestra de la versatilidad de nuestra propuesta.

En su presentación ante expertos de diferentes ámbitos, por lo general Itacio ha suscitado críticas muy positivas, se ha reconocido la originalidad de sus aportaciones y se ha apreciado la utilidad del método, que viene a ocupar parte de un vacío considerable en la tecnología de componentes.

## 6.3. Trabajo futuro

La propuesta que Itacio representa constituye en sí misma un paso adelante, pero un método básico como este abre simultáneamente multitud de posibles líneas de investigación y desarrollo. Asimismo, las limitaciones que se han asumido en esta disertación con el fin de centrar el trabajo de investigación dentro de límites manejables pueden también abordarse en etapas futuras.

- El propio modelo Itacio admite diversas mejoras que podrían abordarse en el futuro.
  - **Gestión del conocimiento**. Sería muy conveniente incorporar técnicas del campo de la Ingeniería del Conocimiento. Estos trabajos incluirían el estudio de vías que permitan mejorar la homogeneidad de las bases de conocimiento, su eficiencia y su corrección, desarrollando si es necesario métodos de análisis y diseño específicos para recoger los requisitos y garantías de los componentes.
  - **Corrección de las especificaciones**. Una línea de investigación interesante sería precisar métodos para que las expresiones restrictivas de los componentes se ajustaran al comportamiento real de los mismos, y que este hecho fuese demostrable. Esto requeriría adaptar las técnicas de desarrollo formal o de programación a partir de especificaciones a la idiosincrasia de Itacio y siempre bajo los condicionantes de simplicidad y facilidad de uso.
  - **Inferencias probabilísticas o aproximadas**. En esta disertación se entiende como una ventaja el diagnóstico inequívoco de la corrección de un sistema. No obstante, en algunos entornos o situaciones podría ser interesante realizar verificaciones cuyo resultado no fuese binario (correcto o incorrecto) sino probabilístico; el modelo Itacio podría enriquecerse con ideas de lógica difusa o razonamiento aproximado para dar cabida a este tipo de diagnósticos.
  - **Relajación de las condiciones de corrección topológica**. Antes de poder realizar la verificación, los criterios de corrección topológica establecidos para el modelo Itacio exigen que los puntos de conexión de todos los componentes se encuentren conectados. En ocasiones, sería interesante contemplar la verificación parcial, para que pudiese aplicarse a un sistema en construcción que cuenta con fuentes o sumideros aún sin conectar. Seguramente bastaría con tener en cuenta esta posibilidad, definiendo en el modelo el comportamiento de elementos *terminadores* ficticios o algún artefacto similar que permitiese la ejecución del proceso de inferencia en estas condiciones.
- La relación de Itacio con la Ingeniería del software también abre muchas vías de trabajo.
  - **Herramientas de producción**. Tras las lecciones aprendidas con los prototipos, el primer paso obvio sería la construcción de un sistema basado en Itacio y destinado a producción, es decir, una herramienta de verificación profesional. Esto incluiría como primer paso facilidades de edición gráfica e interfaz con el usuario, pero incluiría también trabajo en todas las demás áreas del sistema.

- **Integración con procesos de desarrollo**. En esta disertación el modelo Itacio se presenta aplicado a casos concretos y eso da idea del papel que puede jugar, pero de cara a la adopción de la técnica sería importante estudiar con detenimiento la relación adecuada entre este método y el proceso de desarrollo: vías de integración de Itacio con herramientas convencionales de análisis / diseño / programación, incorporación del modelo a diversas metodologías y procesos de desarrollo, etc.

- **Nuevos campos de aplicación del modelo**. Evidentemente, los casos de estudio aquí desarrollados no agotan las posibilidades de experimentación. Hay lugar para la búsqueda de nuevas interpretaciones del modelo de componentes que permitan su aplicación en más ámbitos y etapas del proceso de desarrollo de software.

- **Ingeniería inversa como sistema de búsqueda de defectos**. Aplicando ingeniería inversa a código ya existente, de forma que el código fuente se convierta en una trama de microcomponentes o componentes (tales como operadores, funciones de biblioteca, etc.), sería posible descubrir defectos ocultos de forma rigurosa y automatizada, sin invertir nada en la reconstrucción del código bajo un paradigma distinto. Examinar código fuente de forma automática buscando defectos es, como se ha visto, una tarea compleja. Merecería la pena estudiar si la traducción automatizada de la estructura del código fuente a microcomponentes resulta viable, lo que abriría las puertas a la aplicación de Itacio para verificar el sistema en el nivel de instrucciones del código fuente, funciones, objetos, etc.

- **Búsqueda de componentes**. En el capítulo  se han presentado trabajos de otras personas sobre enfoques de especificación formal para buscar en una biblioteca los componentes que satisfagan ciertas necesidades. Basándose en las expresiones restrictivas que los acompañan en Itacio, quizás sería posible realizar esas búsquedas "por inferencia"; al igual que en la verificación se relacionan los requisitos de un componente con las garantías de aquel al que está conectado, en este caso se trataría de poner en relación las restricciones y garantías de los componentes de la biblioteca con las restricciones y garantías que el usuario plantea.

- **Anticipación y ayuda al diseño**. Una línea de investigación particularmente interesante consistiría en, dado un sistema incompleto, examinar la base de conocimientos que lo describe, deduciendo (de forma automática) por qué no se satisfacen los requisitos que estén sin satisfacer. Esto equivaldría a "generar las reglas que hacen falta" en la base de conocimientos para que esta describa un sistema correcto. A partir de esto, y enlazando con el punto anterior, el sistema sería capaz de sugerir qué componentes deben incluirse en ciertas partes del sistema, completando el diseño por sí solo.

- **Relación entre expresiones restrictivas y código fuente**. Las expresiones restrictivas se redactan de manera relativamente independiente del código fuente (ni siquiera es seguro que en un caso de aplicación de Itacio esté implicado de forma directa ningún código fuente). Esto puede tener como consecuencia que las expresiones restrictivas *mientan* sobre el comportamiento

real del código fuente. Un campo de actuación sería poner en relación métodos de análisis de programas para que las expresiones restrictivas se *dedujesen* del código fuente y se garantizase la coherencia entre ambos.

- Existen algunas líneas de trabajo relacionadas con ciertas técnicas formales que podría resultar interesante explorar.
  - **Especificación de la semántica de las restricciones y del sistema de composición de restricciones mediante la semántica monádica reutilizable.** Esto permitiría la obtención de manera automática de un sistema de verificación de restricciones (en [Lb01] puede verse cómo se obtiene intérpretes a partir de estas especificaciones). Este sistema podría acoplarse a las herramientas desarrolladas para Itacio, con la ventaja de que dicho sistema sería altamente flexible. Se podría modificar, de forma modular, la semántica del modelo de especificación / verificación, y esto tendría inmediato reflejo en la implementación, de modo que la herramienta podría adaptarse con gran facilidad a diferentes variaciones o ampliaciones de Itacio para necesidades concretas.
  - **Definición de diferentes aspectos de los componentes gracias al sistema de especificación modular**. Se podría desarrollar un sistema de especificación semántica modular que contemplara diversos aspectos de los componentes: versiones, dependencias temporales, características funcionales, etc. Puesto que la semántica de un lenguaje de programación se puede describir mediante mónadas, sería razonable pensar en el mismo sistema como expresión de la semántica de las operaciones de los componentes. Esta posibilidad se ha evaluado en capítulos anteriores de esta tesis, y se ha descartado porque requeriría ciertos avances en el estado del arte (lo que iba contra algunos de nuestros objetivos), pero con el desarrollo adecuado podría ofrecer ventajas. De hecho, a efectos prácticos muchos de los desarrollos realizados en este campo se han implementado en lenguajes de programación lógica como Prolog.
  - **Creación de un lenguaje independiente y extensible de propósito específico** para la descripción y verificación de las interacciones entre componentes. Los lenguajes de propósito específico (como los ADLs) frecuentemente tienen el inconveniente de que limitan el campo de acción de su usuario cuando afronta situaciones imprevistas; pero en este caso, la posibilidad de incorporar nuevos aspectos semánticos al lenguaje (en virtud de las facilidades de composición que ofrece la semántica monádica reutilizable) es una ventaja, ya que permitiría su extensión a medida que surgiesen nuevas necesidades.
  - **Adaptación de una técnica de especificación semántica al trabajo con componentes.** La semántica de acción es un intento de proporcionar una especificación semántica más legible; sería interesante estudiar el modo de combinar la semántica de acción con la semántica monádica reutilizable, desarrollando mecanismos para describir componentes software, con lo que ganaría mucho en idoneidad para su aplicación en este ámbito (se ganaría en

legibilidad y en modularidad respecto a otras técnicas de especificación semántica).

# 7. Bibliografía


[Ab96] Ahmed Abd-Allah. *Composing Heterogeneous Software Architectures*. Doctoral Dissertation, Center for Software Engineering, University of Southern California, Agosto de 1996.

http://sunset.usc.edu/TechRpts/dissertation.html

[AB92] L. M. Alonso, J. Bermúdez. *Métodos formales de diseño de programas: VDM*. Facultad de Informática, Universidad del País Vasco, Enero de 1992. FISS-D-53.1-LSI-92

[AB95] Ahmed Abd-Allah, Barry Boehm. *Reasoning about the Composition of Heterogeneous Architectures*. USC Technical Report USC-CSE-95-503, Center for Software Engineering, Computer Science Department, University of Southern California, 1995.

[ABLE] Sitio web del Proyecto ABLE en Carnegie Mellon.

http://www.cs.cmu.edu/Web/Groups/able

[Ac99] Franz Achermann. *Piccola White Paper*. Software Composition Group. Draft: 6 de diciembre de 1999. Disponible en-línea en:

http://iamwww.unibe.ch/~scg/Research/Piccola/white.pdf

[ADLTK] Towards an ADL Toolkit (sitio web del proyecto).

http://www-2.cs.cmu.edu/~acme/adltk/index.html

[Aesop] Sitio web del proyecto Aesop.

http://www.cs.cmu.edu/Web/Groups/able/aesop

[AG97] Ken Arnold, James Gosling. *The Java Programming Language*. Addison-Wesley, diciembre de 1997. ISBN: 0201310066

[Ai00] B. Aiken et.al. *Network Policy and Services: A Report of a Workshop on Middleware*. RFC 2768, febrero de 2000.

[Al97] Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. Thesis, Carnegie Mellon University. CMU Technical Report CMU-CS-97-144, Mayo de 1997.

[Alpha] Sitio web de AlphaWorks (IBM). http://www.alphaworks.ibm.com

[ALSN00] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz. *Piccola – a Small Composition Language*. Software Composition Group, Institute of

Computer Science and Applied Mathematics, Universidad de Berna, Suiza (Febrero de 2000). Disponible en-línea en:

http://www.iam.unibe.ch/~scg/Research/Piccola/

[AN01] Franz Achermann, Oscar Nierstrasz. *Applications = Components + Scripts — A Tour of Piccola*. Software Architectures and Component Technology, Mehmet Aksit (Editor), Kluwer, 2001. Disponible en-línea en:

http://www.iam.unibe.ch/~scg/Research/Piccola/tour.pdf

[ANSI78] Computer and Business Equipment Manufacturers Association (editores). *Programming Language FORTRAN*. American National Standard Institute, Inc., 1978. X3.9-1978, revisión de ANSI X3.9-1966.

[Ar96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5

[BC89] Kent Beck y Ward Cunningham. *A laboratory for teaching object-oriented thinking*. Actas de OOPSLA'89 (Object-Oriented Programming Systems, Languages and Applications), Special Issue of SIGPLAN Notices, vol. 24, núm. 10, págs. 1-6, New Orleans, LA, Octubre de 1989. ACM.

[BE93] A. Birrel, D. Evers, G. Nelson, S. Owicki, E. Wobber. *Distributed garbage collection for network objects*. Technical Report 116, DEC Systems Research Center, Palo Alto (1993).

[BeC85] Jean-François Bergeretti, Bernard A. Carré. *Information-Flow and Data-Flow Analysis of while-Programs*. ACM Transactions on Programming Languages and Systems, Vol. 7, Nº 1, Enero de 1985, págs. 37-61.

[Be84] Magín Berenguer. *Arte en Asturias*. Caja de Ahorros de Asturias y El Comercio, S.A. ISBN: 84-7925-006-2.

[BH94] Jonathan P. Bowen, Michael G. Hinchey. *Ten Commandments of Formal Methods*. Oxford University Computing Laboratory Technical Monograph, 1994. También publicado como Jonathan P. Bowen. *Ten commandments of formal methods*. IEEE Computer, abril de 1995.

[BH94b] Jonathan P. Bowen, Michael G. Hinchey. *Seven More Myths of Formal Methods*. Technical Report PRG-TR-7-94, Oxford University Programming Research Group, Junio de 1994.

[BHS91] Ferenc Belina, Dieter Hogrefe, Amardeo Sarma. *SDL with applications from protocol specification*. Prentice-Hall International, 1991.

[BJRR98] Grady Booch, Ivar Jacobson, James Rumbaugh, Jim Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, octubre de 1998. ISBN: 0-201-57168-4.

[BN84] A. D. Birrel, B. J. Nelson. *Implementing Remote Procedure Calls*. ACM Transactions on Computer Systems, 2(1), págs. 39-59.

[Bo94] Grady Booch. *Object-Oriented Analysis and Design with Applications, 2 nd edition*. Addison-Wesley Object Technology Series, February 1994. ISBN: 0805353402.

Edición española: *Análisis y diseño orientado a objetos con aplicaciones, segunda edición*. Addison-Wesley / Díaz de Santos, 1996. ISBN: 0-201-60122-2

[BR87] Ted Biggerstaff, Charles Richter. *Reusability framework, assessment and directions*. IEEE Software, págs. 41-49, Julio de 1987.

[Br93] Kraig Brockschmidt. *Inside OLE. Second Edition*. Redmond, WA: Microsoft Press, 1993. (Development Library, Books and Periodicals).

[Br96] Kraig Brockschmidt . *What OLE Is Really About*. Technical Article / OLE, julio de 1996 (disponible en Microsoft Developer Network, octubre de 1999).

[BS92] Barry Boehm, William Scherlis. *Megaprogramming*. Proceedings of Software Technology Conference, DARPA. ARPA, 1992.

[BS97] Martin Büchi, Emil Sekerinski. *Formal Methods for Component Software: The Refinement Calculus Perspective*. Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP '97). Turku Centre for Computer Science, TUCS General Publication nº 5, ISBN 952-12-0039-1. Septiembre de 1997.

[Bw97] Jonathan P. Bowen. *B-hold the Future of Software Development*. The Times Higher Education Supplement, Multimedia computer books, 1267:30, 14 de Febrero de 1997. Revisión de [Ar96]. Disponible en-línea en:

http://www.afm.sbu.ac.uk/people/jpb/publications/thes-b.html

[BZ83] Daniel Brand, Pitro Zafiropulo. *On communicating finite-state machines*. Journal of the. ACM, vol 30, nº 2, Abril de 1983, págs. 323-342.

[CA00] Catalysis Resource Center, de Computer Associates. Catalysis Benefits in Lay Terms. Computer Associates International, Inc, 2000. Disponible en-línea en:

http://www.platinum.com/consult/crc/benefits.htm

[CC76] Patrick Cousot, Radhia Cousot. *Static Determination of Dynamic Properties of Programs*. Proceedings of the 2nd International Symposium on Programming. Editor: B. Robinet. París, 13-15 de abril de 1976.

[CC77] Patrick Cousot, Radhia Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. Conference Record of the Fourth ACM Symposium on Principles of Programming Languages. Los Angeles, California, 17-19 de Enero de 1977.

[CC01] Patrick Cousot, Radhia Cousot. *Compositional Separate Modular Static Analysis of Programs by Abstract Interpretation*. Proceedings of the Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet, SSGRR 2001, Compact disc, L'Aquila, Roma, Italia, 6-12 de Agosto de 2001.

[Ce01] Agustín Cernuda del Río. *Diagnóstico remoto de configuración de componentes software en Windows*. Póster presentado en las Jornadas de Transferencia de Tecnología de la Universidad de Oviedo (13-14 de septiembre de 2001). Publicado en el Catálogo de Oferta Tecnológica de la Universidad de Oviedo (Vicerrectorado de Investigación), 2001.

[Ce01b] Agustín Cernuda del Río. *LoadSpy: Profiling Program Loading*. Windows Developer's Journal, Vol. 12, nº 2, Febrero de 2001. CMP Media Inc., San Francisco, EEUU. ISSN: 1083-9887. http://www.wdj.com

[CF99] A. Cortesi, G. Filé (editores). *Static Analysis*. Proceedings of 6th International Symposium (SAS'99), Venecia, Italia. Springer, LNCS 1694, septiembre de 1999.

[CFPTV01] C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, A. Vallecillo. *Extending CORBA Interfaces with Protocols*. The Computer Journal, Vol. 44, nº 5, 2001, págs. 448-462.

[CH78] Patrick Cousot, Nicolas Halbwachs. *Automatic Discovery of Linear Restraints Among Variables of a Program*. Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages. Tucson, Arizona, 23-25 de Enero de 1978.

[Ck94] S. Cook, J. Daniels. *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Englewood Cliffs, N. J. : Prentice Hall, 1994.

[CLC00] Agustín Cernuda del Río, José Emilio Labra Gayo, Juan Manuel Cueva Lovelle. *Itacio: A Component Model for Verifying Software at Construction Time*. III ICSE Workshop on CBSE. 22nd International Conference on Software Engineering. 5-6 de Junio de 2000, Limerick, Irlanda.

http://www.sei.cmu.edu/cbs/cbse2000/papers/index.html

[CLC01a] Agustín Cernuda del Río, José Emilio Labra Gayo, Juan Manuel Cueva Lovelle. *A Model for Integrating Knowledge into Component-Based Software Development*. Proceedings 4th International ICSC Symposium - Soft Computing and Intelligent Systems for Industry. ICSC Academic Press, 26-29 de Junio de 2001. ISBN: 3-906454-27-4

http://www.icsc-naiso.org/conferences/sois2001/index.html.

[CLC01b] Agustín Cernuda del Río, José Emilio Labra Gayo, Juan Manuel Cueva Lovelle. *Verificación y validación mediante un modelo de componentes*. Actas del Simposio Iberoamericano de Sistemas de Información e Ingeniería del Software en la Sociedad del Conocimiento (SISOFT-2001), Bogotá (Colombia), 29-31 de Agosto de 2001.

[CLC01c] Agustín Cernuda del Río, José Emilio Labra Gayo, Juan Manuel Cueva Lovelle. *Verifying Reuse Contracts with a Component Model*. VI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2001), Almagro, Ciudad Real, 21-23 de noviembre 2001.

[CLC01d] Agustín Cernuda del Río, José Emilio Labra Gayo, Juan Manuel Cueva Lovelle. *Applying the Itacio Verification Model to a Component-Based Real-Time Sound Processing System*. Workshop on Constraint Logic Programming and Software Engineering (CLPSE), Seventeenth International Conference on Logic Programming. Pafos, Chipre, 31 de diciembre de 2001.

http://www.utdallas.edu/~gupta/clpse

http://www.cs.ucy.ac.cy/~iclpcp01/

[CM81] W.F. Clocksin, C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[Cm93] D. Coleman et al. *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, N. J. : Prentice Hall, 1993.

[Co96] Patrick Cousot. *Abstract Interpretation*. ACM Computing Surveys, Vol. 28, Nº 2, Junio de 1996.

[Co96b] Patric Cousot. *Program Analysis: The Abstract Interpretation Perspective*. ACM Computing Surveys, Vol. 28, Nº 4es.

[CSPOx] The CSP Archive, Universidad de Oxford.

http://archive.comlab.ox.ac.uk/csp.html

[CW97] Edmund M. Clarke, Jeannette Marie Wing. *Formal Methods: State of the Art and Future Directions*. ACM, 1997.

[DAE] Sitio web del proyecto Daedalus.

http://www.di.ens.fr/~cousot/projects/DAEDALUS/index.shtml

[DCE98] The Open Group. *DCE Today*. Julio de 1998, ISBN: 1-85912-157-8

[DGH79] R. O. Duda, J. G. Gasching, P. E. Hart. *Model design in the PROSPECTOR consultant system for mineral exploration*. En Expert Systems in the Micro-Electronic Age (D. Michie, editor). Edinburgh University Press, 1979.

[DS99] *Microsoft DirectX Media SDK / DirectShow*. Microsoft Corporation, 1999. Incluído en Microsoft Developer Network Library, Octubre de 1999.

[EC99] Sitio Web de ECLiPSe. http://www.icparc.ic.ac.uk/eclipse

[Ei99] Nancy Eickelmann. Entrevista a David Lorge Parnas. Software Engineering Notes, vol. 24 nº 3, ACM Press, Mayo de 1999.

[EK76] van Emden, M.H y Kowalski, Robert. *The Semantics of Predicate Logic as a Programming Language*. Journal of the ACM, volumen 23, nº 4, Octubre de 1976, págs. 733-742.

[EN86] Uffe Engberg, Mogens Nielsen. *A Calculus of Communicating Systems with Label-passing*, Report DAIMI PB-208, Computer Science Department. Universidad de Aarhus, 1986.

[Fr93] T. Frühwirt et al. *Constraint Logic Programming - An Informal Introduction*. Technical report ECRC-93-5. European Computer-Industry Research Centre, Febrero 1993.

[FO75] L. D. Fosdick, L. J. Osterweil. *Validation and global optimization of programs*. En Proceedings of the 4th Texas Conference on Computing Systems (Austin, Texas) patrocinada por IEEE Computer Society, 1975.

[Ga95] Cristina Gacek. *Exploiting Domain Architectures in Software Reuse*. ACM-SIGSOFT SSR'95, 1995.

[Ga97] Cristina Gacek. *Detecting Architectural Mismatches During Systems Composition*. Technical Report USC-CSE-97-TR-506. Qualifying Report for partial fulfillment of Computer Science Department requirements, Center for Software Engineering (University of Southern California), 8 de Julio de 1997.

[GACB95] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, Barry Boehm. *On the Definition of Software System Architecture*. ICSE 17 Software Architecture Workshop. Abril de 1995.

[GAO95] David Garlan, Robert Allen, John Ockerbloom. *Architectural Mismatch or Why it's hard to build systems out of existing parts*. Proceedings of the 17th International Conference on Software Engineering, Abril de 1995.

[GB98] Cristina Gacek, Barry Boehm. *Composing Components: How Does One Detect Potential Architectural Mismatches?* Position paper to the OMG-DARPA-MCC Workshop on Compositional Software. Center for Software Engineering (University of Southern California). Enero de 1998.

http://sunset.usc.edu/TechRpts/Papers/usccse98-505.html

[Gl86] R. J. van Glabbeek. *Notes on the methodology of CCS and CSP*. Theoretical Computer Science 177(2), 1997, págs. 329-349.

[GM96] Joseph A. Goguen, Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations in Computer Science, MIT Press, 1996.

[Go98] Peggi Goodwin. *COM, ActiveX Controls, and Microsoft Windows CE*. Marzo de 1998 (disponible en Microsoft Developer Network, octubre de 1999).

[Gold] Editor de audio GoldWave, http://www.goldwave.com

[Gr95] David Garlan. *An Introduction to the Aesop System*. School of Computer Science, Carnegie Mellon University. 11 de Julio de 1995. Disponible en-línea en:

http://www-2.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps

[GS93] David Garlan, Mary Shaw. *An introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., 1993.

[Ha00] Pat Hall. *Educational Case Study – what is the model for an ideal component? Must it be an object?* III ICSE Workshop on CBSE. 22nd International Conference on Software Engineering. 5-6 de Junio de 2000, Limerick, Irlanda.

http://www.sei.cmu.edu/cbs/cbse2000/papers/index.html

[Hl90] J. A. Hall. *Seven Myths of Formal Methods*. IEEE Software, 7(5):11-19, Septiembre de 1990.

[HLS97] Koen De Hondt, Carine Lucas, Patrick Steyaert. *Reuse Contracts as Component Interface Descriptions*. Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP '97). Turku Centre for Computer Science, TUCS General Publication nº 5, ISBN 952-12-0039-1. Septiembre de 1997.

[HMW01] Dick Hamlet, Dave Mason, and Denise Woit. *Theory of Software Component Reliability*. Actas - 23rd International Conference on Software Engineering (ICSE'2001), Mayo de 2001 (Toronto, Canadá).

[Ho92] Ian Holland. *The Design and Representation of Object-Oriented Components*. Tesis doctoral, College of Computer Science, Northeastern University, 1992.

[Hoa69] C. A. R. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, vol. 12, nº 10, págs. 576-580. Octubre de 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 0-13-15327-8

[Hr95] Markus Hortsmann. *From CPP to COM*. Octubre de 1995. Recogido en la Microsoft Developer Network (MSDN) Library, Julio 1999 (y anteriores) en el apartado Technical Articles/Component Object Model/Component Object Model.

[IN88] INMOS Limited. *occam2 Reference Manual*. Prentice-Hall, 1988. ISBN: 0-13-629312-3

[It01] Sitio web ITPapers.com. http://itpapers.com

[Ja94] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, marzo de 1994. ISBN: ISBN: 0201544350.

[Ja99] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, enero de 1999. ISBN: 0201571692.

[JG88] Geraint Jones, Michael Goldsmith. *Programming in occam2*. Prentice-Hall, 1988. ISBN: 0-13-730334-3.

[JN94] Neil D. Jones, Flemming Nielsen. *Abstract Interpretation: a Semantics-Based Tool for Program Analysis*. 30 de Junio de 1994.

[JSGB00] Bill Joy, Guy Steele, James Gosling y Gilad Bracha. *The Java Language Specification. Addison-Wesley*, 2nd edition, 2000.

[KCAB97] R. Kazman, P. Clements, G. Abowd, L. Bass. *Classifying Architectural Elements as a Foundation for Mechanism Matching*. Proceedings of COMPSAC 1997, Washington DC, Agosto de 1997.

[Ki73] G. A. Kildall. *A Unified Approach to Global Program Optimisation*. En Conference Record of the first ACM Symposium on Principles of Programming Languages, págs. 194-206. ACM Press.

[KJB00] Zeynep Kiziltan, Torsten Jonsson, Brahim Hnich. *On the Definition of Concepts in Component Based Software Development*. Uppsala University Department of Information Science, Febrero de 2000. Disponible en-línea en:

http://citeseer.nj.nec.com/437569.html

[KK93] E. Koutsofios, S. C. King. *Drawing graphs with dot*. AT&T Bell Laboratories, Murray Hill, NJ, 1993

[Kl99] Onno Kluyt. *JavaBeans Technology: Unlocking The BeanContext API*. Mayo de 1999. Disponible en

http://developer.java.sun.com/developer/technicalArticles/index.html

[Ko00] Cris Kobryn. *Modeling Components and Frameworks with UML*. Communications of the ACM, Vol. 43 nº 10, Octubre de 2000.

[KSW99] John Kauffman, Kevin Spencer, Thearon Willis. *Beginning ASP Databases*. Wrox Press Inc, Septiembre de 1999. ISBN: 1861002726

[Kyma] Kyma, un producto de Symbolic Sound Corporation.

http://www.symbolicsound.com

[La93] John Lamping. *Typing the specialization interface*. Proceedings of OOPSLA'93 (26 de Septiembre al 1 de Octubre, Washington DC, USA), volumen 28(10) de ACM Sigplan Notices, págs. 201-214. ACM Press, Octubre de 1993.

[Lau00] Kung Kiu Lau. *The Role of Logic Programming in Next-generation Component-based Software Development*. Proceedings of Workshop on Logic Programming and Software Enginering, Londres, Julio de 2000 (G. Gupta y I. V. Ramakrishnan, editores).

[LAN00] Markus Lumpe, Franz Achermann, Oscar Nierstrasz. *A Formal Language for Composition*. Foundations of Component Based Systems, Gary Leavens y Murali Sitaraman (Editores), págs. 69—90, Cambridge University Press, 2000. Disponible en-línea en:

http://www.iam.unibe.ch/~scg/Archive/Papers/Lump00aFormalLanguage.pdf

[Lb99] Jose Emilio Labra Gayo. Proyecto Docente, perfil de Lógica y Programación Lógica y Funcional. Oviedo, Junio de 1999.

[Lb01] Jose Emilio Labra Gayo. *Desarrollo modular de procesadores de lenguajes a partir de especificaciones semánticas reutilizables*. Tesis Doctoral, Departamento de Informática, Universidad de Oviedo, 2001.

[Lc96] David C. Luckham. *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events*. 29 de Agosto de 1996.

[Lc98] David C. Luckham. *Rapide: A Language and Toolset for Causal Event Modeling of Distributed System Architectures*. Proceedings of the International Conference on Worldwide Computing and Its Applications (WWCA '98), Tsukuba, Japón, 4 y 5 de Marzo de 1998. Lecture Notes in Computer Science, Vol. 1368, Springer, 1998. ISBN: 3-540-64216-1. págs. 88-96.

[LCLC01] Jose Emilio Labra Gayo, Juan Manuel Cueva Lovelle, María Cándida Luengo Díez, Agustín Cernuda del Río. *Specification of logic programming languages from reusable semantic building blocks*. En International Workshop on Functional and (Constraint) Logic Programming, nº 2017, Kiel (Alemania), septiembre de 2001. Christian-Albrechts-Universität Kiel.

[LCLG01] Jose Emilio Labra Gayo, Juan Manuel Cueva Lovelle, María Cándida Luengo Díez, Bernardo Martín González. *A language prototyping tool based on semantic building blocks*. En Eight International Conference on Computer Aided Systems Theory and Technology (EUROCAST '01), vol. 2178 de Lecture Notes in Computer Science, Las Palmas de Gran Canaria, Febrero de 2001.

[Le97] Christian Lenz Cesar. *Graph theoretical foundation classes and a framework for graph drawing and layout*. Disponible en

http://www.dfki.uni-kl.de/km/java/java/gfc/doc/

[LF98] Jose Emilio Labra, Ana Isabel Fernández. *Lógica proposicional para informática*. Área de lenguajes y sistemas informáticos, Departamento de informática,

Escuela Universitaria de Ingeniería Técnica de Informática de Oviedo. Universidad de Oviedo, Noviembre de 1998.

[LH96] Sheng Liang y Paul Hudak. *Modular denotational semantics for compiler construction*. En Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping, vol. 1058 de Lecture Notes of Computer Science, págs. 219-234. Springer-Verlag, 1996.

[LHJ95] Sheng Liang, Paul Hudak y Mark P. Jones. *Monad transformers and modular interpreters*. En 22nd ACM Symposium on Principles of Programming Languages, San Francisco (EEUU). ACM, Enero de 1995.

[LHR88] Karl J. Lieberherr, Ian Holland, Arthur J. Riel. *Object-Oriented programming: An objective sense of style*. En Object-Oriented Programming Systems, Languages and Applications Conference, en Special Issue of SIGPLAN Notices, págs. 323-334, San Diego, CA., Septiembre de 1988. Se puede encontrar una versión corta en IEEE Computer, Junio de 1988, sección Open Channel, págs. 78-79.

[Li96] J. L. Lions. *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*. París, 19 de Julio de 1996.

http://www.esrin.esa.it/htdocs/tidc/Press/ Press96/ariane5rep.html

[LLCC01] Jose Emilio Labra Gayo, María Cándida Luengo Díez, Juan Manuel Cueva Lovelle, Agustín Cernuda del Río. *LPS : A language prototyping system using modular monadic semantics*. En Electronic Notes in Theoretical Computer Science, vol. 44, Mark van den Brand y Didier Parigot (editores), Elsevier Science Publishers, 2001.

[LPT94] Peter Gorm Larsen, Nico Plat, Hans Toetenel. *A Formal Semantics for Data Flow Diagrams*. Formal Aspects of Computing, 6, 1994.

[LR92] David Long, Dan Ruder. *Introduction to Microsoft Windows Dynamic-Link Libraries*. Microsoft Developer Network (MSDN) Library Archive. Agosto de 1992.

[Lu97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. Tesis doctoral, Vrije Universiteit Brussel, Bélgica. Septiembre de 1997.

[LVC89] Mark A. Linton, John M. Vlissides, Paul R. Calder. *Composing User Interfaces using InterViews*. IEEE Computer Magazine, págs. 8-22, Febrero de 1989.

[LW98] Gary T. Leavens, Jeannette Marie Wing. *Protective Interface Specifications*. TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France. Volume 1214 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pages 520-534.

[Ma99] Florian Martin. *Generating Program Analyzers*. Dissertation Zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften (Dr.-Ing.) der Technischen Fakultät der Universität des Saarlandes. Junio de 1999.

[Mc96] Steve McConnell. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, Julio de 1996. ISBN: 1556159005.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, Jeff Kramer. *Specifying Distributed Software Architectures*. Proceedings of the 5th European Software Engineering

Conf. (ESEC 95), Sitges, Barcelona, Septiembre de 1995. LNCS 989, (Springer-Verlag), 1995, págs. 137-153.

[Me99] Bertrand Meyer. *Construcción de software orientado a objetos*. Prentice Hall, 1999. ISBN: 84-8322-040-7. (Versión española de *Object-Oriented Software Construction*. Prentice Hall, 1988. Segunda edición: ISBN: 0136291554, Abril de 1997).

[Mercury] Sitio web del proyecto Mercury.

http://www.cs.mu.oz.au/research/mercury/index.html

[Mi96] Microsoft Corporation. *DCOM Technical Overview*. Microsoft Developer Network Library, Noviembre de 1996.

[Mi97] Microsoft Corporation. *Microsoft Personal Web Server Documentation*, documentación en línea que acompaña a Microsoft Personal Web Server 5.0, 1997.

[Ml89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Ml93] Robin Milner. *The polyadic pi-calculus: a tutorial*. En F. L. Hamer, W. Hrauer y H. Schwichtenberg, editores, *Logic and Algebra of Specification*. Springer-Verlag, 1993. También como informe ECS-LFCS-98-180, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1991.

[Ml99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge Books, junio de 1999. ISBN: 0-521-64320-1.

[Mo94] Carroll Morgan. *Programming from Specifications, 2nd Edition*. Prentice Hall International, Londres, 1994. ISBN 0-13-123274-6. Revisión de la segunda edición (1998) disponible en-línea en formato HTML y PostScript:

http://users.comlab.ox.ac.uk/carroll.morgan/PfS/

[MPW89] Robin Milner, Joachim Parrow, David Walker. *A calculus of mobile processes*. Informes ECS-LFCS-89-85 y ECS-LFCS-89-86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.

[MSW97] Tobias Murer, Daniel Scherer, Andy Würtz. *Improving component interoperability*. En Special Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP'96, Linz. (Editor: Max Mühlhäuser). Págs. 150-158. Dpunkt Verlag, 1997.

[Ms92] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[MT98] Nenad Medvidovic, Richard N. Taylor. *Separating Fact from Fiction in Software Architecture*. Proceedings of the 3rd International Software Architecture Workshop, págs. 105-108, Noviembre de 1998.

[NACO97] Gleb Naumovich, George S. Avrunin, Lori A. Clarke, Leon J. Osterweil. *Applying Static Analysis to Software Architectures*. Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), 1997.

[NASA93] NASA Office of Safety and Mission Assurance. *Software Formal Inspections Guidebook*. Documento NASA-GB-A302, aprobado en Agosto de 1993. Disponible en-línea en :

http://swg.jpl.nasa.gov/resources (versión texto)

http://satc.gsfc.nasa.gov/fi (versiones texto y PDF)

[NM95] Oscar Nierstrasz, Theo Dirk Meijler. *Requirements for a Composition Language*. En *Object-Based Models and Languages for Concurrent Systems*, Paolo Ciancarini, Oscar Nierstrasz y Akinori Yonezawa (Editores), págs. 147—161. Springer-Verlag, 1995.

[NNH99] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN: 3-540-65410-0

[OCL] Sitio web de OCL en IBM.

http://www-4.ibm.com/software/ad/library/standards/ocl.html.

[OMG] Sitio web del Object Management Group. http://www.omg.org

[OMG97] *The Common Object Request Broker: Architecture and Specification, Revision 2.0* (julio de 1995, actualización de julio de 1996). Object Management Group, formal document 97-02-25.

[OMG97b] *A Discussion of the Object Management Architecture*. Object Management Group, formal document 00-06-41. Enero de 1997.

[OS98] Open Systems Resources, Inc. *Windows NT Virtual Memory System (Part I)*. The NT Insider, Issue 2 Vol. 5 (Marzo / Abril de 1998).

http://www.osr.com/ntinsider/1998/Virtualmem1/virtualmem1.htm

[OSF] The Open Software Foundation (The Open Group).

http://www.opengroup.org

[Pa96] David Parnas. *Mathematical methods: What we need and don't need*. En *An Invitation to Formal Methods*, IEEE Computer, Abril de 1996.

[PAG] Sitio web de PAG : The Program Analyzer Generator.

http://rw4.cs.uni-sb.de/~martin/pag/

[Pe98] Charles Petzold. *Programming Windows, Fifth Edition*. Microsoft Press, Noviembre de 1998. ISBN: 1-57231-995-X.

[PI98] President's Information Technology Advisory Committee (PITAC) Interim Report to the President, agosto de 1998.

[Pi95] Benjamin C. Pierce. *Foundational Calculi for Programming Languages*. *CRC Handbook of Computer Science and Engineering*. 22 de diciembre de 1995.

[Piccola] Sitio web del Software Composition Group de la Universidad de Berna sobre Piccola.

http://iamwww.unibe.ch/~scg/Research/Piccola/

[Pl81] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Technical report DAIMI FN-19, Computer Science Department, Aarhus University (Dinamarca), 1981.

[Pr92] Roger S. Pressman. *Software Engineering*. McGraw-Hill, 1992.

[Po99] Jeff Prosise. *Programming Windows with MFC, 2nd edition*. Microsoft Press, 13 de mayo de 1999. ISBN: 1572316950.

[PT97] Benjamin C. Pierce, David N. Turner. *Pict: A Programming Language Based on the Pi-Calculus*. CSCI Technical Report #475, Indiana University, marzo de 1997.

[Ra97] Rational, Microsoft, HP, Oracle et al. *Object Constraint Language Specification, version 1.1*. Septiembre de 1997.

http://www-4.ibm.com/software/ad/standards/ad970808_UML11_OCL.pdf

[Rapide] Sitio web de Rapide en la Universidad de Stanford.

http://pavg.stanford.edu/rapide/

[Rb00] John Robbins. Entrevista en Developer Network Journal, nº 19, Julio/Agosto de 2000. Microsoft Corporation, publicado por Matt Publishing Limited, Bristol, Reino Unido.

http://www.dnjonline.com

[RMRR98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, David S. Rosenblum. *Integrating Architecture Description Languages with a Standard Design Method*. Proceedings of the The 20th International Conference on Software Engineering (ICSE'98), 19-25 de Abril, Kyoto (Japón). IEEE, 1998.

[Ro92] Dale Rogerson. *Exporting with Class*. Microsoft Developer Network Library Archive, Technical Articles, C/C++ Articles. Octubre 1992.

[Ro97] Dale Rogerson. Inside COM. Microsoft Press, Redmond, Washington, febrero de 1997. ISBN: 1572313498

[Ro00] Mike Rosen. *The Software in the Middle*. Software Magazine (softwaremag.com), febrero de 2000. Disponible en línea en:

http://www.softwaremag.com/archive/2000feb/MRosen.html.

[Rs97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997. ISBN 0-13-674409-5. Sitio web del libro:

http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency.

[Ru91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenson. *Object-Oriented Modelling and Design*. Prentice Hall, enero de 1991. ISBN: 0136298419.

[Ru98] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, diciembre de 1998. ISBN: 020130998X.

[RWA96] Tryvge Reenskaug, Per Wold, Odd Arild Lehne. *Working with Objects: The OOram Software Engineering Method*. Addison-Wesley/Manning, octubre de 1996. ISBN 1-884777-10-4.

[Sa85] M. Shaw. *The Carnegie-Mellon Curriculum for Undergraduate Computer Science*. Springer-Verlag, 1985.

[Sc99] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley & Sons, noviembre de 1999. ISBN: 0471623733.

[SCG] Sitio web del Software Composition Group, Universidad de Berna.

http://www.iam.unibe.ch/~scg/

[SDK95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. *Abstractions for Software Architecture and Tools to Support Them*. IEEE Transactions on Software Engineering, vol. 21, nº 4, págs. 314-335, abril de 1995.

[Seresco] Sitio web de Seresco, S.A. http://www.seresco.es

[SG93] Mary Shaw, David Garlan. *An Introduction to Software Architecture*. Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Co., 1993.

[SG95] Mary Shaw, David Garlan. *Formulations and Formalisms in Software Architecture*, Volume 1000, Springer-Verlag Lecture Notes in Computer Science, 1995.

[SG96] Mary Shaw, David Garlan. *Software Architectures – Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[SGJ95] Archana Shankar, Dr. David Gilbert, Michael Jampel. *Transient Analysis of Linear Circuits Using Constraint Logic Programming*. TCU/CS/1995/17, Department of Computer Science, School of Informatics, City University, London. Noviembre de 1995.

http://www.soi.city.ac.uk/~drg/publications/techreports/drg_95_17.html

[Sh79] E. H. Shortliffe. *Computer-Based Medical Consultations: Mycin*. New York, American-Elsevier, 1979.

[Si97] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997. ISBN 0-9660176-3-3. Disponible en http://www.dspguide.com/pdfbook.htm

[SLMH96] Patrick Steyaert, Carine Lucas, Kim Mens, Theo D'Hondt. *Reuse contracts: Managing the evolution of reusable assets*. Proceedings of OOPSLA'96 (del 6 al 10 de Octubre, San Jose, California), volumen 31(10) de ACM Sigplan Notices, págs. 268-285. ACM Press, 1996.

También disponible en http://progwww.vub.ac.be/prog/pools/rcs

[Sm95] Raymond M. Smullyan. *First-Order Logic*. Dover Pubns, Febrero 1995. ISBN: 0486683702.

[Sp89] J. M. Spivey. *An introduction to Z and formal specifications*. Software Engineering Journal, Enero de 1989.

[Sp92] J. M. Spivey. *The Z Notation*. Prentice Hall, 1992.

(Descatalogado; la referencia de la notación Z se encuentra en http://spivey.oriel.ox.ac.uk/~mike/zrm/index.html)

[St67] C. Strachey. *Fundamental Concepts in Programming Languages*. Higher Order and Symbolic Computation, 13 de abril de 2000. Reimpreso de Lecture Notes, International Summer School in Computer Programming at Copenhagen, 1967.

[STM] STMicroelectronics (fabricante actual del Transputer y otros productos de la desaparecida INMOS): http://eu.st.com/stonline/index.shtml

[Su97] Graham Hamilton (editor). *JavaBeans 1.01 Specification*. Sun Microsystems, julio de 1997. http://java.sun.com/beans

[SW99] Desmond Francis D'Souza, Alan Cameron Wills. *Objects, Components and Frameworks with UML : The Catalysis℠ Approach*. Object Technology Series – Addison-Wesley, 1999. ISBN: 0-201-31012-0.

[SWa01] Davide Sangiorgi, David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[Syma] Sitio web de Symantec, Inc. http://www.symantec.com

[Sz97] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997 (reimpreso en 1998).

ISBN: 0-201-17888-5

[Th00] François Thomasset. Carta privada, 24 de Julio de 2000.

[Thom] Sitio web de François Thomasset. http://www-rocq.inria.fr/~thomasse

[TP86] Borland International, Inc. *Turbo Pascal 3 User Manual*. Septiembre de 1986. ISBN 0-87524-003-8

[Tu93] Kenneth J. Turner. *The formal description techniques: an introduction to Estelle, Lotos and SDL*. John Wiley and Sons, 1993.

[UML] Sitio web de UML. http://www.uml.org.

[UniCon] Sitio web del proyecto UniCon.

http://www-2.cs.cmu.edu/afs/cs/project/vit/www/unicon/index.html

[Vi97] Steve Vinoski. *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments*. IEEE Communications Magazine. Febrero de 1997.

http://www.cs.wustl.edu/~schmidt/PDF/vinoski.pdf

[VML] Web Workshop - Specs & Standards - VML Overview. Documentación en línea de Microsoft. http://msdn.microsoft.com/workshop/author/vml

[VNC] Sitio web de VNC. http://www.uk.research.att.com/vnc/index.html

[Wa83] D. Warren. *An Abstract Prolog Instruction Set*. Technical Note 309, SRI, Octubre de 1983.

[Waves] Waves (KS Waves, Ltd.). http://www.waves.com

[WaveX] Proyecto WaveX. http://www.agustincernuda.com/wavex_esp.html

[Wi95] Jeannette Marie Wing. *Hints to Specifiers*. Artículo CMU-CS-95-118R, School of Computer Science, Carnegie-Mellon University. Mayo de 1995.

[WK99] Jos Warmer, Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, marzo de 1999. ISBN: 0-201-37940-6.

[Wl97] Mark Wallace et al. *ECLiPSe: A Platform for Constraint Logic Programming*. William Peney Laboratory, Imperial College, Londres. Agosto 1997.

[WP00] Kurt C. Wallnau, D. Plakosh. WaterBeans: A Custom Component Model and Framework. III ICSE Workshop on CBSE. 22nd International Conference on Software Engineering. 5-6 de Junio de 2000, Limerick, Irlanda.

http://www.sei.cmu.edu/cbs/cbse2000/papers/index.html

[Ws92] Allan Cameron Wills. *Formal Methods applied to Object-Oriented Programming*. Tesis Doctoral, Universidad de Manchester (1992).

[Wt96] David A. Watt. *Why don't programming language designers use formal methods?* En Anais XXIII Seminário Integrado de Software e Hardware (ed. R. Barros), págs. 1-16, Universidade Federál de Pernambuco, Recife, Brasil, 1996.

[WW89] Rebecca Wirfs-Brock y Brian Wilkerson. *Object-oriented design: A responsibility-driven approach*. En Object-Oriented Programming Systems, Languages and Applications Conference, Special Issue of SIGPLAN Notices, págs. 71-76, New Orleans, LA, 1989. ACM.

[WWW90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[YS97] Daniel M. Yellin, Robert E. Strom. *Protocol Specifications and Component Adaptors*. ACM Transactions on Programming Languages and Systems, Vol. 19, Nº 2, Marzo de 1997, págs. 292-333.

[ZW97] Amy Moormann Zaremski, Jeannette Marie Wing. *Specification Matching of Software Components*. ACM Transactions on Software Engineering and Methodology, Vol. 6 No. 4, págs. 333-369, Octubre de 1997.

[ZW98] Amy Moormann Zaremski, Jeannette Marie Wing. *Signature Matching: A Key to Reuse*. School of Computer Science, Carnegie Mellon University, 1998.

# 8. Índice

**H**



**I**



**J**



**K**



**L**



**M**