Lección 2
Métodos Formales para Sistemas Abiertos

Antonio Vallecillo
Dept. Lenguajes y Ciencias de la Computación. Universidad de Málaga.
ETSI Informática. Campus Teatinos, s/n. 29071 Málaga, Spain.
av@lcc.uma.es

1 Introducción

Si ya la programación tradicional se enfrenta a serias dificultades para tratar con las características específicas de los sistemas abiertos y distribuidos, los métodos formales no se encuentran en mejor situación. Por un lado existe la necesidad de razonar formalmente sobre las aplicaciones desarrolladas, y ser capaces no sólo de verificar que satisfacen las especificaciones del usuario, sino de poder demostrar ciertas propiedades sobre ellas (de seguridad o de viveza). Pero por otro lado, los problemas que aparecen en este tipo de sistemas complican mucho su tratamiento desde un punto de vista formal:

1. En primer lugar, las extensiones independientes de los sistemas y la composición tardía de componentes restringen bastante el tipo de propiedades que pueden probarse sobre los sistemas y los componentes.

2. Las aplicaciones se desarrollan en este tipo de sistemas en base a componentes reutilizables, y por tanto las propiedades de las aplicaciones han de poder derivarse de las de éstos, lo que suele limitar el tipo de propiedades que pueden probarse.

3. La evolución de los componentes ha de poder contemplarse también desde un punto de vista formal, así como la reconfiguración dinámica de las aplicaciones conforme los elementos del sistema cambien.

4. Y por último, la falta de visión global del sistema, la dispersión de sus componentes y la posibilidad de errores y retrasos en las comunicaciones no son fáciles de tratar desde una perspectiva formal.

Actualmente se trabaja en tres frentes principales dentro de los métodos formales para sistemas abiertos y distribuidos: la adecuación y extensión de los modelos tradicionales, la definición de nuevos modelos, y en la combinación de ellos.

En las siguientes secciones describiremos varios de los principales modelos y notaciones utilizados actualmente en el tratamiento formal de los sistemas abiertos y distribuidos, así como de algunos modelos que combinan distintos formalismos.

2 Un poco de historia

Todo comenzó con [Floyd, 1967], que introdujo el concepto de corrección para programas secuenciales, e introdujo un método para probarlo (corrección parcial) basado en pre-condiciones, post-condiciones, y condiciones de terminación. El método consiste en anotar en cada punto de control de un programa un predicado lógico, que debe ser cierto cuando la ejecución del programa se encuentre en ese punto.

Basado en ese método, [Hoare, 1967] construye una lógica en donde las fórmulas son de la forma $\{P\}S\{Q\}$, siendo P y Q los predicados que establecen las pre y post-condiciones del programa S. De esta forma, si el predicado P es cierto antes de la ejecución de S, entonces el predicado Q también será cierto después de su ejecución. Hoare definió también una serie de reglas de inferencia para poder reducir la demostración de la corrección de un programa a la demostración de la corrección de cada una de las sentencias que lo componen.

Floyd y Hoare cambiaron la forma en la que hasta entonces se pensaba en los programas, viéndolos como transformadores de estados, en vez de como generadores de eventos. El concepto de *estado* de un programa se convirtió en el eje central para la demostración de la corrección de programas secuenciales.

[Ashcroft, 1975] fue el primero en extender el razonamiento sobre programas basado en estados para tratar también programas concurrentes. Ashcroft generalizó el método de Floyd, expresando la concurrencia mediante operaciones *fork* y *join*. Los predicados que se anotan en cada punto de control del programa pasan a ser *invariantes*, que deben verificarse en ese punto independientemente del orden en que se ejecuten el resto de instrucciones.

Por otro lado, [Owicki y Gries, 1976] generalizaron el método de Hoare para poder razonar sobre programas concurrentes. La concurrencia se introduce mediante las claúsulas **cobegin-coend**, y añadieron a la lógica de Hoare la regla siguiente:

$$\frac{\{P_1\}S_1\{Q_1\}, \ldots, \{P_n\}S_n\{Q_n\}}{\{P_1 \wedge \ldots \wedge P_n\}\ \textbf{cobegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{coend}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

(siempre que $\{P_1\}S_1\{Q_1\}, \ldots, \{P_n\}S_n\{Q_n\}$ no interfieran entre sí).

Los métodos de Ashcroft y Owicki-Gries están muy relacionados entre sí [Lamport, 1993], y su importancia reside en generalizar el concepto de corrección de programas para considerar el concepto de *invariante*: en vez de razonar sobre lo que es cierto o deja de serlo antes y después de la ejecución del programa, se pasa a razonar sobre lo que es cierto a lo largo de su ejecución.

Basado en el método de Owicki y Gries se han construido métodos para razonar sobre lenguajes más sofisticados, como puede ser CSP, para el que [Apt et al., 1980] y [Levin y Gries, 1981] desarrollaron de forma independiente dos métodos formales de razonamiento. El primero de esos equipos, liderado por de Roever, también han construido métodos para más lenguajes, incluido hasta un subconjunto del propio Ada [Gerth y de Roever, 1984]. Sin embargo, el problema de estos métodos es que normalmente son muy buenos para programas simples, pero inútiles para programas grandes puesto que se dispara su complejidad (tanto algorítmica, como de manejabilidad y comprensión).

3 Lógica temporal

El primer avance serio que se realizó sobre los dos métodos anteriores lo introdujo [Pnueli, 1977] con el uso de la lógica temporal para razonar sobre programas concurrentes.

La lógica original de Pnueli es una extensión de la lógica proposicional clásica, en donde el tiempo es discreto y lineal, y las fórmulas se construyen con proposiciones evaluables en los distintos estados por los que atraviesa un programa. Cada uno de esos estados constituye un *mundo*, frente al único mundo en donde se evalúa la lógica proposicional clásica. El operador *siempre* ($\Box$) comprueba la validez de una fórmula en todos los estados futuros por los que atravesará un programa.

La semántica de esta lógica se define en base a estados, en donde un estado es una asignación de valores a las variables de un programa. Una secuencia infinita de estados $(s_0, s_1, \ldots)$ se denomina *comportamiento*, y representa la ejecución de un programa; la terminación se representa como una repetición infinita del estado final del programa. El significado $[\![P]\!]$ de un predicado lógico P es una función que asocia a cada estado s de un programa un valor lógico, que se obtiene al sustituir las variables libres de P por el valor que tienen en ese estado s. Por otro lado, el significado $[\![F]\!]$ de una fórmula F es una función que asocia a cada comportamiento $\sigma = (s_0, s_1, \ldots)$ de un programa un valor lógico, definido por:

$$[\![P]\!](s_0, s_1, \ldots) == [\![P]\!](s_0), \quad \text{para todo predicado lógico } P$$

$$[\![F \diamond G]\!](s_0, s_1, \ldots) == [\![F]\!](s_0, s_1, \ldots) \diamond [\![G]\!](s_0, s_1, \ldots), \text{ para todo operador booleano } \diamond$$

$$[\![\Box F]\!](s_0, s_1, \ldots) == \forall\, n \in \mathbb{N} \bullet [\![F]\!](s_n, s_{n+1}, \ldots)$$

Intuitivamente, una fórmula es una afirmación sobre el comportamiento de un programa a partir de un instante de tiempo dado (en esta lógica temporal se define la variable *ahora*). La fórmula $\Box F$ afirma que F es cierto y siempre lo será. Además de $\Box$ se definen más operadores, como pueden ser *alguna vez* ($\Diamond$) o *conlleva* ($\rightsquigarrow$). La fórmula $\Diamond F$ se define como $\neg\, \Box \neg\, F$, y en esta lógica significa que o la fórmula F es cierta ahora, o lo será alguna vez en el futuro. Por otro lado, la fórmula $F \rightsquigarrow G$ se define como $\Box(F \Rightarrow \Diamond G)$, e indica que si F es alguna vez cierta, entonces G también lo será, o bien en ese momento, o bien más tarde.

Para aplicar la lógica temporal a los programas lo que se hace es definir el significado $[\![\Pi]\!]$ de un programa Π como un conjunto de comportamientos. Con esto, diremos que un programa Π satisface una fórmula F ($\Pi \models F$) si $[\![F]\!](\sigma)$ es cierto para todos los comportamientos σ de $[\![\Pi]\!]$. El razonamiento basado en los *invariantes* de un programa puede entonces escribirse mediante la siguiente *regla de invariancia* [Lamport, 1993]:

$$\frac{\forall\, S \text{ operación atómica de } \Pi \bullet \{I\}S\{I\}}{\Pi \models I \Rightarrow \Box I}$$

De esta forma es posible ver a los métodos de Ashcroft y Owicki-Gries como casos particulares de aplicación de esta regla. Aunque esta formulación es bastante agradable y simple, la regla de invariancia no proporciona demasiada ayuda a la hora de demostrar propiedades de invariancia.

Donde sí demuestra la lógica temporal su utilidad es para la demostración de propiedades de seguridad (*safety*) y viveza (*liveness*). Las propiedades de seguridad afirman que nada malo pasará, mientras que las de viveza afirman que algo bueno terminará pasando. Estas propiedades se suelen expresar mediante el operador $\leadsto$.

Antes de hablar de estas propiedades de seguridad y viveza hay que hablar también sobre las propiedades de imparcialidad (*fairness*), en sus dos variantes: débil y fuerte. La imparcialidad débil sobre una operación atómica S de un programa Π afirma que si S siempre está lista para ser ejecutada, tarde o temprano terminará ejecutándose. La imparcialidad fuerte afirma que si S está repetidamente lista para ser ejecutada (aunque se desactive también repetidamente), tarde o temprano terminará ejecutándose. Estos requisitos pueden expresarse mediante las dos siguientes reglas:

$$\frac{P \Rightarrow (S \text{ activo}), \{P\}S\{Q\}}{\Pi \models (\Box P) \leadsto Q} \quad \text{(imparcialidad débil)}$$

$$\frac{P \Rightarrow (S \text{ activo}), \{P\}S\{Q\}}{\Pi \models (\Box\Diamond P) \leadsto Q} \quad \text{(imparcialidad fuerte)}$$

La lógica temporal permite la integración de las propiedades de invarianza en demostraciones de viveza, utilizando la siguiente regla:

$$\frac{\Pi \models P \leadsto Q, \Pi \models Q \Rightarrow \Box Q}{\Pi \models P \leadsto \Box Q}$$

Una de las ventajas de la lógica temporal es que permite la demostración de propiedades de viveza de forma práctica: en [Owicki y Lamport, 1982] se demuestra que para probar $F \leadsto G$ basta con encontrar una colección de fórmulas $\mathcal{H}$ que contengan a F y probar, para cada una de esas fórmulas H de $\mathcal{H}$, que $\Pi \models H \leadsto (J \vee G)$, para algún $J \prec H$.

La importancia de las propiedades de seguridad y viveza se pone también de manifiesto tras el trabajo de [Alpern y Schneider, 1985]. Ellos definieron el que un comportamiento finito ρ satisfaga una fórmula F: ρ satisface a F si ρ puede ser extendido a un comportamiento infinito que satisfaga F. Con esto, una propiedad se puede considerar de seguridad si el que sea cierta para un comportamiento σ es equivalente a que sea cierta para todo prefijo finito suyo. Y una propiedad es de viveza si y sólo si es satisfecha por todo comportamiento finito de un programa. [Alpern y Schneider, 1985] probaron entonces que toda fórmula temporal puede expresarse mediante una combinación de una propiedad de seguridad y otra de viveza.

La lógica temporal admite numerosas extensiones y variantes para trabajar con distintos tipos de tiempos (lineal, ramificado, discreto, continuo, con pasado, con intervalos, etc.), y con distintos operadores adicionales, como *siguiente* ($\bigcirc$), que indica que una fórmula es válida en el siguiente estado al estado actual, o los operadores de pasado *hasta ahora* ($\boxminus$), *justo antes* ($\ominus$) y *previamente* ($\diamondminus$). Estos tres son los duales a los operadores de futuro $\Box$, $\bigcirc$ y $\Diamond$, e implican la validez de una fórmula en todos los estados pasados, sólo en el anterior, o en alguno de los anteriores, respectivamente.

[Moszkowski, 1986] es un buen libro que describe la semántica precisa de los operadores más comunes con los que se trabaja en lógica temporal.

3.1 Unity y CC++

Chandy y Misra reconocieron la importancia de los invariantes para razonar sobre programas concurrentes, aunque también pensaron que la mayor fuente de confusión en el método de Owicki-Gries es el control del flujo de los programas. Para solucionar esta situación elaboraron un lenguaje de programación puramente paralelo y sin secuencias para el flujo de control, al que denominaron Unity [Chandy y Misra, 1988].

Todos los programas Unity tienen la misma estructura, que puede ser expresada en su forma más simple en términos del **do** de Dijkstra como:

$$\textbf{do } P_1 \rightarrow S_1 \,[\!]\, \ldots \,[\!]\, P_n \rightarrow S_n \textbf{ od}$$

La imparcialidad de los programas Unity la proporciona la forma en la que se seleccionan las sentencias con guardas activas dentro de una cláusula **do**.

Para razonar sobre los programas, Chandy y Misra desarrollaron una lógica que puede ser considerada como un subconjunto de la lógica temporal, basada en las fórmulas $\Box P$ y $P \leadsto Q$, y que fundamentalmente utiliza los siguientes operadores adicionales:

P **unless** $Q == \Box((P \wedge \neg\, Q) \Rightarrow \bigcirc(P \vee Q))$

stable $P == P$ **unless** *false* $\quad [\Box(P \Rightarrow \bigcirc P)]$

invariant $P == (\boxminus P) \wedge (\Box P)$

constant $P == (\textbf{stable } P) \wedge (\textbf{stable } \neg\, P)$

P **ensures** $Q == (P \textbf{ unless } Q) \wedge \Diamond((P \wedge \neg\, Q) \Rightarrow \bigcirc Q)$

Las cuatro primera fórmulas son de seguridad, mientras que P **ensures** Q, junto con $P \leadsto Q$, son de viveza. A partir de ellas y algunas derivadas suyas, Chandy y Misra proponen una de las formas más elegantes de razonar sobre los programas concurrentes.

Además de para razonar sobre programas concurrentes, Misra ha tratado de utilizar Unity para especificarlos [Misra, 1990]. Sin embargo, Unity no es lo suficientemente expresivo para dicha tarea, pues se necesita añadir variables auxiliares a la especificación que, una vez añadidas, no se distinguen de las variables 'reales' del propio programa. El hecho de que las variables auxiliares no tengan por qué implementarse hace que Unity sea considerado solamente como un método semi-formal para especificar programas concurrentes [Lamport, 1993].

Con posterioridad a Unity, [Chandy y Kesselman, 1992] proponen un método para razonar sobre programas concurrentes en sistemas abiertos, y que aplican para los programas desarrollados con el lenguaje composicional CC++, también basado en representar el comportamiento de los programas como secuencias de estados. Una *acción* se define entonces como una relación binaria sobre estados, lo que intuitivamente corresponde a la idea de que una acción hace transitar el sistema de un estado a otro. Una acción $\mathcal{A}$ es *ejecutable* en un estado s si existe otro estado s' tal que $(s, s') \in \mathcal{A}$. A partir de las acciones, Chandy y Kesselmann definen el concepto de *computación* de un programa (o proceso), como un estado inicial s_0 y una secuencia de pares $(\mathcal{A}_i, s_i)_{i>0}$, en donde cada acción $\mathcal{A}_i$ lleva el programa del estado s_{i-1} a s_i, y que satisface la siguiente regla de imparcialidad: Si C es una computación infinita y $\mathcal{A}$ una acción ejecutable en un punto de la computación, entonces existe un punto posterior en la computación en donde se ejecuta $\mathcal{A}$, o $\mathcal{A}$ ya no es ejecutable.

En general, una propiedad P de un programa Π en un sistema abierto se define como un predicado sobre todas las computaciones de $\Pi \parallel \Gamma$, para *cualquier* otro proceso arbitrario Γ. El problema de esta definición es que, por ser muy general, reduce notablemente el tipo de propiedades que se pueden probar sobre un programa. [Chandy y Kesselman, 1992] ofrecen una solución muy elegante para este problema, basándose en probar propiedades para $\Pi \parallel \Gamma$, no siendo Γ un proceso arbitrario, sino aquel que tenga una interfaz que sea *propia* para Π. La idea es estudiar las propiedades de la composición de programas sólo a partir de sus interfaces, y sólo para aquellos en los que tenga sentido componerlos. Esta es la idea clave y fundamental para la demostración de propiedades en sistemas abiertos, y seguida por numerosos autores, entre los que destacaremos [Duke et al., 1996]. Sin embargo, el único problema que presenta el trabajo de Chandy y Misra es que restringen demasiado las características que deben verificar las interfaces para ser *propias*, restringiendo por tanto el tipo de programas que pueden componerse con otros para razonar sobre ellos, y el tipo de propiedades que pueden probarse.

3.2 TLA

Apoyándose en la notación matemática habitual y en la lógica temporal aparece TLA [Lamport, 1991], acrónimo de *Temporal Logic of Actions*, otro lenguaje para describir el comportamiento de los programas concurrentes y razonar sobre ellos.

TLA añade a la notación matemática clásica tres operadores: $'$, $\Box$ y $\exists\!\!\exists$. El primero de ellos permite distinguir entre los valores de las variables antes y después de la ejecución de una sentencia. $\Box$ es el operador *siempre* de la lógica temporal clásica, y $\exists\!\!\exists$ es un cuantificador temporal, que se diferencia del cuantificador existencial $\exists$ en que el primero afirma la existencia de una serie de valores —uno para cada estado de un comportamiento—, en vez de afirmar la existencia de un solo valor, como hace el segundo operador. Por lo demás, $\exists\!\!\exists$ obedece a las reglas habituales del cálculo de predicados para la cuantificación existencial.

En TLA los programas se describen mediante una fórmula, que tiene tres partes: una condición inicial (*Init*) que debe cumplir el estado inicial del programa, una relación $\mathcal{N}$ que establece las acciones posibles del programa como transiciones entre estados, y un conjunto L de fórmulas que establecen la *imparcialidad* del programa:

$$\Pi == Init \wedge \Box[\mathcal{N}]_v \wedge L$$

En esta fórmula, v representa el conjunto de variables relevantes a cada una de las acciones del conjunto $\mathcal{N}$, y L viene descrito por conjunciones de fórmulas que establecen que tarde o temprano las acciones del programa que estén habilitadas se ejecutarán. Como puede observarse, TLA es muy similar al método de Chandy y Kesselman citado anteriormente.

En TLA, como en matemáticas, no hay distinción entre programas, especificaciones y propiedades. Todo se expresa mediante fórmulas, con lo cual se evitan los 'saltos' que hay que dar entre unos niveles y otros, debido fundamentalmente a los diferentes lenguajes en los que se expresan, y a que cada uno de esos lenguajes tiene un nivel de expresividad distinto. Todo esto se simplifica en TLA debido a la uniformidad con la que se expresan todos los conceptos.

Cara a la especificación de programas, Lamport ha definido TLA+, un lenguaje que completa TLA y aporta estructuración para el desarrollo modular de programas grandes. El problema fundamental de este método es la falta de conexión entre TLA y el lenguaje de implementación concreto, puesto que no se dispone de una metodología clara para derivar programas a partir de especificaciones, ni para 'elevar' implementaciones al nivel de la especificación para comprobar su corrección.

4 Especificaciones axiomáticas

Ya desde el comienzo de la década de los 80 se vio que no bastaba con probar la invariancia de los programas y sus propiedades de seguridad o progreso: también es preciso poder expresar otros requisitos más complejos, como pueden ser las políticas de planificación o la selección de recursos, y poder razonar sobre ellos. En general, y de forma intuitiva, una especificación es una descripción abstracta del comportamiento *correcto* que debe observar un programa. El hecho de ser abstracta implica que debe ser independiente de todos aquellos detalles irrelevantes para el usuario del sistema, como puede ser el lenguaje en el que finalmente se implemente o la plataforma en donde sea ejecutado, aunque no por ello debe ignorarse que toda especificación sirve para describir un sistema que finalmente ha de implementarse, y por tanto debe proporcionar mecanismos para comprobar que una implementación *satisface* un especificación, es decir, que la implementación es *correcta*. Por otro lado, *verificar* significa poder probar que una especificación implementa correctamente a otra.

El método axiomático consiste en escribir las especificaciones como listas de propiedades, aquellas que debe satisfacer el programa. Más formalmente, cada propiedad es una fórmula en alguna lógica, y la especificación total es la conjunción de todas las fórmulas. Diremos que una especificación X verifica otra especificación Y si y sólo si las propiedades de Y pueden deducirse a partir de las de X; en otras palabras, la verificación se reduce a implicaciones lógicas.

Expresar una especificación como una lista de propiedades que debe cumplir un programa sin detallar cómo ha de hacerlo es un buen objetivo; sin embargo, el problema se plantea al decidir el lenguaje en el que se han de describir esas propiedades. La lógica temporal aparece como candidato idóneo inicialmente, pues permite expresar de formas natural el tipo de propiedades que necesitamos. Sin embargo, y como hemos discutido antes, la lógica temporal también plantea ciertas desventajas de índole práctica, sobre

todo a la hora de derivar la implementación de una especificación, o de comprobar la corrección de una implementación dada. Aparecen por tanto las especificaciones ecuacionales (o algebraicas), que utilizan ecuaciones para describir el comportamiento de los componentes de los sistemas.

Básicamente una especificación ecuacional es un par (Σ, E), con Σ es un alfabeto que define la sintaxis de las operaciones que se pretenden especificar, y E es un conjunto de ecuaciones que relaciona las operaciones de Σ. Denotaremos por $T_{\Sigma,E}(X)$ al Σ-álgebra con las clases de equivalencia de los términos de Σ, módulo las ecuaciones de E, y con variables en un conjunto dado $X = \{x_1, \ldots, x_n\}$.

De esta forma, las especificaciones ecuacionales permiten describir el comportamiento de un sistema en base a la especificación de los tipos de datos que lo componen y las posibles operaciones que pueden realizarse en él. Como puede observarse, es un enfoque fundamentalmente basado en el uso de Tipos Abstractos de Datos.

Este estilo de especificación basado en ecuaciones tiene dos utilidades principales, para las cuales es especialmente adecuado. En primer lugar, interpretando las ecuaciones en ambos sentidos es posible razonar sobre los sistemas especificados y demostrar propiedades sobre ellos. Y en segundo lugar, interpretando las ecuaciones como reducciones de términos, es posible construir prototipos sobre los sistemas especificados. Dichos prototipos se basan en la reducción de términos hasta conseguir sus formas normales (expresión más simple de sus representantes canónicos). De esta forma, para demostrar una propiedad P sobre un objeto del sistema (dado por un término `t` de tipo `T`) basta con especificar P mediante ecuaciones como una función `P:T->Bool`, y reducir el término `P(t)`. La demostración de propiedades utilizando las ecuaciones bidireccionalmente se basa en la denominada *lógica ecuacional*, mientras que la demostración de propiedades y prototipado rápido de los sistemas mediante la reducción de términos se basa en la denominada *lógica de reescritura*.

4.1 Las lógicas de reescritura y ecuacional

Comenzaremos definiendo una *teoría de reescritura (etiquetada)* $\mathcal{R}$ como un cuarteto $\mathcal{R} = (\Sigma, E, L, R)$, en donde el par (Σ, E) es una especificación, L es un conjunto de etiquetas, y R es un conjunto de pares $R \subseteq L \times T_{\Sigma,E}(X)^2$, compuestos por una etiqueta y un par de clases de equivalencia de los términos de la especificación. A los elementos del conjunto R se les denomina *reglas de reescritura*, y en vez de representarlos en la forma $(r, ([t], [t']))$ utilizaremos la notación $r : [t] \longrightarrow [t']$. Cuando además queramos hacer referencia explícita a las variables $\bar{x} = (x_1, \ldots, x_k)$ que intervienen en la definición de un término, lo notaremos como $t(\bar{x})$ o por $t(x_1, \ldots, x_k)$; finalmente, notaremos por $t(\bar{w}/\bar{x})$ al término que se obtiene al reemplazar las ocurrencias de las variables de $\bar{x}$ por las de $\bar{w}$. A las expresiones $r : [t] \longrightarrow [t']$ se les denomina normalmente *secuentes*.

Dada una teoría de reescritura $\mathcal{R}$, diremos que de $\mathcal{R}$ se deduce el secuente $[t] \longrightarrow [t']$, y notaremos por $\mathcal{R} \vdash [t] \longrightarrow [t']$, si y sólo si ese secuente puede obtenerse aplicando un número finito de veces las siguientes cuatro reglas solamente:

1. **Reflexividad**. Para todo $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}$$

2. **Congruencia**. Para todo $f \in \Sigma_n, n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1], \ldots, [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}$$

3. **Reemplazamiento**. Para cada regla $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ de R,

$$\frac{[w_1] \longrightarrow [w'_1], \ldots, [w_n] \longrightarrow [w'_n]}{[t(\bar{w}/\bar{x})] \longrightarrow [t(\bar{w}'/\bar{x})]}$$

4. **Transitividad**. Para todo $[t_1], [t_2], [t_3] \in T_{\Sigma,E}(X)$

$$\frac{[t_1] \longrightarrow [t_2], [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

La *lógica de reescritura* es aquella que cuenta sólo con esas cuatro reglas básicas. Para definir la *lógica ecuacional* que comentábamos antes, se le añade la regla adicional:

5. **Simetría**. Para todo $[t_1], [t_2] \in T_{\Sigma,E}(X)$,

$$\frac{[t_1] \longrightarrow [t_2]}{[t_2] \longrightarrow [t_1]}$$

Con esta nueva regla, los secuentes que se derivan en esta lógica son *bidireccionales*, y se suelen denotar por $r : [t_1] \leftrightarrow [t_2]$.

4.2 Maude

Existen diferentes lenguajes para especificar sistemas de forma ecuacional, es decir, utilizando solamente lógica de reescritura. Entre los existentes se encuentran OBJ, Maude, EqLog y FOOPS, aunque quizá sea Maude [Meseguer, 2000] el de más ambiciosa cobertura y alcance. Para ilustrar el estilo de Maude especificaremos los números naturales en este lenguaje:

```
fmod NAT is
   sort Nat .
   op 0 : -> Nat .
   op s _ : Nat -> Nat .
   op _ + _ : Nat Nat -> Nat [comm] .
   vars N M : Nat .
   eq N + 0 = N .
   eq N + s M = s (N + M) .
endfm
```

Como puede observarse, un módulo contiene una especificación que define un tipo (`Nat`) y unas operaciones sobre él (cero, sucesor y suma), junto con unas ecuaciones que describen la semántica de esas operaciones. Maude está soportado por lógica ecuacional de pertenencia, que soporta tipos, subtipos, sobrecarga de operadores, y parcialidad. Maude es un lenguaje basado en módulos y permite la genericidad, la meta-programación, y la mayoría de los mecanismos tanto de la programación orientada a objetos como de la orientada a componentes (herencia, polimorfismo, concurrencia, etc.).

En Maude, un sistema orientado a objetos se especifica mediante módulos orientados a objetos en los que se declaran las clases y las subclases que forman el sistema. Una clase se declara en Maude como "`class` C | $a_1 : S_1, ..., a_n : S_n$", donde C es el nombre de la clase, a_i son los nombres de los atributos, y S_i son sus correspondientes tipos. Con esto, un objeto de la clase C es una estructura de la forma `<` $O : C \mid a_1 : v_1, ..., a_n : v_n$ `>`, donde O es el nombre del objeto, y v_i son los valores actuales de los atributos. Los objetos pueden interactuar de diversas formas en Maude, entre ellas mediante mensajes.

En un sistema orientado a objetos y concurrente, el estado se llama su *configuración*, y tiene la estructura de un multiconjunto compuesto por objetos y mensajes que evoluciona mediante reescrituras concurrentes que describen los efectos de los *eventos de comunicación* entre los objetos y los mensajes que intervienen en dichas comunicaciones. La forma general de tales reglas de rescritura es la siguiente:

$$\begin{array}{l} crl\ [r]\ : \\ \quad M_1 \ldots M_m \\ \quad < O_1 : C_1 \mid atts_1 > ... < O_n : C_n \mid atts_n > \\ \longrightarrow \\ \quad < O_{i_1} : C'_{i_1} \mid atts'_{i_1} > ... < O_{i_k} : C'_{i_k} \mid atts'_{i_k} > \\ \quad < Q_1 : D_1 \mid atts''_1 > ... < Q_p : D_p \mid atts''_p > \\ \quad M'_1 \ldots M'_q \\ if\ \ Cond\ . \end{array}$$

en donde r es la etiqueta de la regla, M_i son mensajes, O_i y Q_j son identificadores de objetos, C_i, ${C'}_j$ y D_h son clases, $i_1, \ldots, i_k$ es un subconjunto de $1 \ldots n$, y $Cond$ es una condición booleana (la 'guarda' de la regla). El resultado de aplicar dicha regla es que:

- los mensajes $M_1 \ldots M_m$ desaparecen, es decir, son consumidos;
- el estado, y posiblemente las clases de los objetos $O_{i_1}, \ldots, O_{i_k}$ puede cambiar;

- el resto de objetos O_j desaparecen;
- se crean los nuevos objetos $Q_1, \ldots, Q_p$; y
- se crean los nuevos mensajes $L'_1 \ldots M'_q$, es decir, son enviados.

Cuando varios objetos o mensajes aparecen en la parte izquierda de una regla, necesitan sincronizarse para que se dispare dicha regla. Es por este motivo por lo este tipo de reglas se denominan *síncronas*, mientras que las reglas con un solo objeto y un mensaje en su parte izquierda se denominan *asíncronas*.

Maude proporciona herencia mediante su estructura de tipos ordenada. Así, una declaración de subclase `C < C'` es un caso particular de una declaración de subtipo `C < C'`, mediante la cual todos los atributos, mensajes y reglas de la superclase, así como los nuevos atributos, mensajes y reglas de la subclase caracterizan su estructura y comportamiento. Maude también soporta herencia múltiple [Meseguer, 1993].

A modo de ejemplo, el siguiente módulo especifica un objeto que modela una cuenta bancaria con dos servicios, uno para depositar dinero y otro para sacarlo:

```
(omod BANKACCOUNT is
   protecting MACHINE-INT .
   class Account | bal : MachineInt .
   msgs deposit withdraw : Oid MachineInt -> Msg .
   var A : Oid .
   vars B M : MachineInt .

   rl [deposit]:
     < A : Account | bal : B >
     deposit (A, M)
     => < A : Account | bal : B + M > .

   crl [withdraw]:
     < A : Account | bal : B >
     withdraw (A, M)
     => < A : Account | bal : B - M >
     if M <= B .
endom)
```

Disponer de este tipo de lenguajes presenta numerosas ventajas:

1. En primer lugar, al estar basados en una lógica muy básica, son muy expresivos, pues cualquier otra lógica cumple las cuatro reglas mencionadas antes. De esta forma es posible expresar la mayor parte de los conceptos y requisitos de las aplicaciones.
2. En segundo lugar, Maude es un lenguaje ejecutable, por lo que es posible realizar prototipado rápido de las aplicaciones especificadas con él. Además, la tasa de reescrituras que se está consiguiendo con los últimos compiladores de Maude lo convierten en un lenguaje que puede casi competir con los intérpretes de lenguajes secuenciales.
3. Las especificaciones escritas en Maude pueden también utilizarse para demostrar propiedades. Por ejemplo, la suma se especifica como operación conmutativa en el módulo `NAT`, pero utilizando inducción estructural es fácil también probar la asociatividad de esa operación. Para esto es suficiente considerar las ecuaciones dentro de una lógica ecuacional (es decir, añadir la quinta regla) y usar la lógica proposicional clásica.
4. En Maude las ecuaciones se evalúan todas en paralelo, y el no-determinismo viene impuesto por la elección aleatoria que realiza la máquina de Maude de entre todas las ecuaciones que puede aplicar en un momento dado para reducir un término. Estos hechos hacen que la concurrencia se trate de forma natural en Maude, y que no interfiera con otros mecanismos como puede ser la herencia. Así en Maude el problema de la anomalía de la herencia no se produce.

Frente a estas grandes ventajas, las especificaciones ecuacionales presentan el problema de su demasiado bajo nivel, lo que hace que la especificación de cualquier aplicación no trivial se convierta en difícil de manejar por su extensión.

5 Especificaciones conjuntistas

Otro enfoque para construir especificaciones es el que parte de la teoría de conjuntos, y construye sobre ella una notación capaz de representar y modelar aplicaciones software. De esta forma el punto de partida de estas notaciones son los *tipos abstractos matemáticos* junto con la lógica de predicados para razonar sobre ellos. Z [Spivey, 1992], VDM [Jones, 1994], B [Abrial, 1986] y la familia de lenguajes Larch [Guttag y Horning, 1993] son ejemplos de lenguajes que siguen este enfoque. En esta sección nos centraremos el primero de ellos.

5.1 La notación formal Z

Z es una notación formal basada en la teoría de conjuntos de Zermelo (de ahí la 'Z') que utiliza los conceptos básicos de esa teoría (conjuntos, relaciones, funciones y variables) para describir sistemas y aplicaciones. Nuevos tipos se construyen sobre los anteriores mediante productos cartesianos ($\times$), conjuntos ($\mathbb{P}$), o secuencias (seq). En Z también se pueden introducir nuevos tipos básicos, como por ejemplo [*FECHA*], que representa un tipo denominado *FECHA* cuya estructura no es relevante a ese nivel de la especificación, y por lo tanto se omite. También es posible definir tipos por extensión, como por ejemplo

$$RESULTADO ::= perfecto \mid nocabe \mid noquiero$$

que representa a un tipo enumerado con tres posibles valores.

Sobre estos elementos básicos Z define los denominados *esquemas*, que permiten la construcción estructurada y modular de especificaciones software. En Z los esquemas sirven para modelar los aspectos tanto estáticos como dinámicos de un sistema. Entre los primeros destacamos:

- Los estados que puede alcanzar el sistema.
- Los invariantes que se conservan en todas las transiciones entre estados.

Los esquemas también permiten representar los siguientes aspectos dinámicos de un sistema:

- Las posibles operaciones que se pueden realizar sobre él.
- Las relaciones entre las entradas y salidas del sistema.
- Los cambios de estado del sistema; cuándo y cómo se producen.

Los esquemas se representan mediante cajas que tiene dos partes separadas por una línea horizontal. La superior es la parte de *declaraciones*, en donde se expresa el nombre de las variables y sus tipos (Z es un lenguaje fuertemente tipado). La parte inferior expresa, de forma declarativa, las relaciones entre las variables que forman parte del esquema. Por ejemplo, el siguiente esquema describe una estructura de datos que es una secuencia de números naturales de longitud menor que 10:

ColaAcotada

$$cola : \operatorname{seq} \mathbb{N}$$

$$\#cola \leq 10$$

Para representar operaciones también se utilizan esquemas:

*Incluye*0

$$\Delta ColaAcotada$$
$$num? : \mathbb{N}$$
$$informe! : RESULTADO$$

$$\#cola < 10$$
$$cola' = cola \frown \langle num? \rangle$$
$$informe! = perfecto$$

Este esquema representa una operación que inserta un elemento en la cola, siempre que quepa. Las variables en Z pueden ir *decoradas* con distintos símbolos: $'$ representa a la variable tras la ejecución de

la operación; ? indica que la variable es un parámetro de entrada; y ! indica que es de salida. El operador Δ en la parte de las declaraciones indica que ese esquema modifica las variables de otro. Obsérvese que en la parte inferior de un esquema se describe el efecto de la operación mediante las relaciones entre las distintas variables, pero no cómo se ha de llevar a cabo, es decir, se dice *qué* hace la operación, pero no *cómo* lo hace. Las precondiciones se expresan también como relaciones, y son aquellas que no contienen variables decoradas con $'$ o ! (p.e. $\#cola < 10$). Si una precondición no se satisface, la operación no se realiza.

Z define un cálculo para trabajar con esquemas mediante el cual es posible realizar numerosas operaciones con ellos: conjunción, disyunción, implicación, equivalencia, composición secuencial, etc. De esta forma podemos definir de forma totalmente independiente el esquema

IncluyeConError

$\Xi ColaAcotada$
$num? : \mathbb{N}$
$informe! : RESULTADO$

$\#cola \geq 10$
$informe! = nocabe$

y construir posteriormente con él y con el esquema anterior un esquema compuesto:

$$Incluye \mathrel{\widehat=} Incluye0 \lor IncluyeConError$$

El operador Ξ hace referencia a un esquema cuyas variables no se modifican. Como el sistema de tipos de Z es fuerte, hace falta incluir todos aquellos esquemas cuyas variables se referencien.

La notación Z se sustenta sobre una lógica denominada $\mathcal{W}$, basada en secuentes al estilo de Gentzen (pocos axiomas y muchas reglas de inferencia) [Woodcock y Brien, 1992]. Actualmente $\mathcal{W}$ ha sido extendida a ν [Brien, 1995], aunque el resultado de [Hall y Martin, 1997] permite seguir utilizando esta lógica sin problemas, así como las herramientas existentes para Z construidas sobre $\mathcal{W}$.

- Los secuentes de esta lógica son de la forma $d \mid \Psi \vdash \Phi$, con un *antecedente* $(d \mid \Psi)$ compuesto por una lista de declaraciones d y un conjunto de predicados Ψ, y un *consecuente* compuesto por un conjunto de predicados Φ.

- Las reglas de inferencia para manipular los secuentes son de la forma

$$\frac{\text{premisas}}{\text{conclusión}} \quad [\text{ condición }]$$

  en donde la condición es opcional e indica una pre-condición para para poder aplicar la regla.

- Existen dos meta-funciones interesantes: α devuelve los nombres de las variables declaradas en un esquema, y ϕ los nombres de las variables libres. Por ejemplo, en la siguiente regla de inferencia (para la definición por comprensión de conjuntos) se comprueba que no haya conflictos en los nombres de las variables de t y d:

$$\frac{\vdash \exists\, d \mid P \bullet t = u}{\vdash t \in \{d \mid P \bullet u\}} \quad [\, \phi t \cap \alpha d = \varnothing \,]$$

  En esta regla, d es la declaración, P un predicado, y t y u las expresiones que definen los elementos del conjunto.

La notación Z es muy expresiva para describir las entidades de los sistemas y aplicaciones software, aunque también presenta algunas limitaciones, como las que destaca [Lamport, 1994]: "Z de por sí es inadecuado para especificar sistemas reactivos". Efectivamente, Z es muy adecuado para modelar las entidades estáticas y su comportamiento, pero no para modelar conceptos dinámicos como puede ser el tiempo real o la concurrencia. Sin embargo, [Evans, 1997] ha probado cómo es posible vencer algunas de estas limitaciones, introduciendo genericidad (para especificar comportamiento concurrente en Z), tiempo real, modularidad (para encapsular componentes concurrentes dentro de una especificación Z) y

comunicación síncrona (al estilo de CSP). Algunas de ellas son realmente elegantes y simples, mientras que otras (como la modularidad) se consiguen quizá de una forma ligeramente forzada y antinatural.

Por otro lado, una de las grandes ventajas de Z es que ofrece una metodología y un proceso para la derivación de implementaciones a partir de las especificaciones. Este proceso, denominado *refinamiento* en terminología de Z, viene descrito con mucho detalle por ejemplo en el libro de [Woodcock y Davies, 1996] y ampliado en [Stepney et al., 1998].

5.2 Object-Z

Relacionada con la falta de modularidad de Z se encuentra otra de sus limitaciones: la falta de estructura en las especificaciones, lo que complica mucho aquellas de gran tamaño. En Z todas las variables son globales, y no es posible realizar especificaciones usando una estructura que no sea *plana*.

Una posible solución pasa por introducir la orientación a objetos en Z, y para ello se han barajado fundamentalmente dos posibilidades:

1. utilizar Z siguiendo un estilo orientado a objetos (propuestas de Hall o ZERO); o bien
2. extender Z para construir notaciones realmente orientadas a objetos (MooZ, Object-Z, OOZE, Z++ o ZEST).

Estos enfoques, junto con cada una de las propuestas mencionadas, son discutidos con gran detalle en [Stepney et al., 1992]. En este apartado nos centraremos en una de estas propuestas, Object-Z [Duke et al., 1994].

Object-Z trata de solucionar uno de los inconvenientes de Z, permitiendo agrupar un *estado* junto con un conjunto de *operaciones* sobre ese estado, en el más puro estilo de orientación a objetos. Para ello define el concepto de *clase*, permite declarar objetos de esa clase, así como extender las clases mediante herencia. De esta forma en Object-Z podemos definir la clase:

ColaAcotada

$$cola : \operatorname{seq} \mathbb{N}$$
$$\#cola \leq 10$$

INIT

$$cola = \langle\,\rangle$$

*Incluye*0

$$\Delta(cola)$$
$$num? : \mathbb{N}$$
$$informe! : RESULTADO$$
$$\#cola < 10$$
$$cola' = cola \frown \langle num? \rangle$$
$$informe! = perfecto$$

...

Podemos entonces definir objetos de esa clase como $c : ColaAcotada$, y referirnos a sus operaciones como $c.Incluye0$. Object-Z permite herencia simple, y la declaración $c :\downarrow ColaAcotada$ declara un objeto c de la clase *ColaAcotada* o de cualquiera derivada de ella.

Object-Z incorpora también genericidad en sus clases, y la posibilidad de incluir invariantes de clase, es decir, predicados que se verifican a lo largo de la vida de los objetos de esa clase, y que deben ser respetados por las operaciones de la clase. Los invariantes se expresan como fórmulas en la lógica temporal.

Por otro lado, aparecen operadores especiales en Object-Z para trabajar con las operaciones definidas en las clases, que extienden el cálculo de esquemas de Z. Así aparecen:

- El operador paralelo ($\|$), que se comporta como la conjunción de esquemas ($\wedge$) pero que asocia las variables de entrada y salida del mismo nombre, ocultando las de entrada que consiga emparejar. Es un operador conmutativo y asociativo, y para ello no oculta ninguna variable de salida.

- El operador composición secuencial ($\overset{\circ}{9}$), que se comporta como la composición secuencial de Z, y que realiza la comunicación entre operaciones de izquierda a derecha, asociando las variables de entrada y salida del mismo nombre, y ocultando todas las variables que consiga emparejar. Este operador no es ni asociativo ni conmutativo.

- Por último, el operador de elección ($[]$) selecciona una operación de forma no determinista de entre todas las que satisfagan sus precondiciones. Este operador es asociativo y conmutativo, y no hay comunicación entre operaciones puesto que sólo se ejecuta una.

La lógica que define la semántica axiomática de Object-Z es una extensión de la lógica $\mathcal{W}$ de Z, y se debe a [Smith, 1994]:

- Los secuentes se interpretan ahora en el contexto de una clase, y por tanto adquieren la forma: $A :: d \mid \Psi \vdash \Phi$, siendo A una clase en el sentido de Object-Z.

- Las reglas de inferencia para manipular los secuentes también se ven referidos al contexto de una clase, y el siguiente resultado (*rule-lifting*) permite trabajar con $\mathcal{W}$ dentro de la lógica de Object-Z sin problemas:

  Si la regla $\dfrac{d_1 \mid \Psi_1 \vdash \Phi_1}{d_2 \mid \Psi_2 \vdash \Phi_2}$ [*Cond*] es válida en $\mathcal{W}$,

  entonces la regla $\dfrac{A :: d_1 \mid \Psi_1 \vdash \Phi_1}{A :: d_2 \mid \Psi_2 \vdash \Phi_2}$ [*Cond*] también lo es.

- Por otro lado, la lógica de Z se extiende también para cubrir los nuevos operadores de Object-Z [Smith, 1994].

6 Álgebras de procesos

Otros métodos y notaciones formales para modelar la concurrencia se basan en álgebras de procesos. Aquí la concurrencia se expresa mediante procesos secuenciales que interactúan en paralelo, comunicándose y sincronizándose mediante canales. Sobre estos modelos se razona en base a las posibles trazas de los eventos observables del sistema. Cada proceso se especifica mediante una notación en la que las variables definen los estados, y las constantes o símbolos terminales especifican las acciones, eventos y transiciones. Los sistemas se construyen mediante conjuntos de procesos, que se componen utilizando distintos operadores que modelan la elección, ejecución paralela y comunicación entre los procesos. Estos operadores constituyen un álgebra de procesos.

Milner desarrolló CCS (*Calculus of Concurrent Systems*) [Milner, 1989] como una notación para describir sistemas multiprocesadores y explorar las distintas nociones de equivalencia de procesos. Por otro lado, CSP (*Communicating Sequential Processes*) de [Hoare, 1984] nació como un lenguaje de programación, que luego sirvió de inspiración al lenguaje Occam. Basado en ambos surgió el lenguaje de especificación LOTOS [ISO, 1989b], estándar de ISO para especificar sistemas abiertos.

En este apartado hablaremos del representante más expresivo de este tipo de notaciones formales basadas en álgebras de procesos, el π-cálculo, que surge como extensión de CCS y CSP para expresar los aspectos dinámicos de los sistemas que especifica.

6.1 El π-cálculo

El π-cálculo [Milner et al., 1992] es una notación formal diseñada para la descripción y análisis de sistemas concurrentes cuya topología (conexión entre procesos) sea dinámica y evolutiva. Los sistemas se especifican en π-cálculo como conjuntos de procesos (*agentes*) que interactúan mediante enlaces (*nombres*). Para expresar la evolución dinámica del sistema (*movilidad*) se permite que los nombres de los enlaces puedan pasarse entre unos procesos y otros formando parte de los mensajes que se intercambian [Engberg y Nielsen, 1986]. Cuando un agente recibe un nombre, puede utilizarlo para comunicarse a través de ese enlace, lo que permite una reconfiguración dinámica de la topología del sistema. Este

tratamiento homogéneo de los nombres de los enlaces es lo que hace del π-cálculo un cálculo muy simple, aunque muy expresivo.

En lo que sigue denominaremos $P, Q, \ldots$ a los agentes, $w, x, y, \ldots$ a los nombres de los enlaces, y usaremos tildes ($\tilde{w}$) para referirnos a las secuencias de esos nombres. Con esto, los agentes se construyen en π-cálculo a partir de los siguientes operadores:

$$\mathbf{0} \mid (x)P \mid [x = z]P \mid \tau.P \mid \bar{x}\tilde{y}.P \mid x(\tilde{w}).P \mid P|Q \mid P + Q \mid A(\tilde{w})$$

- En primer lugar, $\mathbf{0}$ expresa un comportamiento no activo por parte del agente. Es el elemento neutro del álgebra.

- En segundo lugar, se pueden utilizar restricciones para crear nombres privados, como en $(x)P$, en donde x es privado al agente P. Otros agentes pueden comunicarse a través de este nombre de enlace, pero es necesario que lo conozcan antes. Para ello P necesita comunicárselo, enviándoselo a través de algún otro enlace como parte de alguno de los mensajes que se intercambian.

- Las comprobaciones (*matchings*) del tipo $[x = z]P$ se comportan como P si x y z son idénticos; si no, se comportan como $\mathbf{0}$.

- Las transiciones silenciosas, expresadas por τ, modelan las acciones internas de los procesos. De esta forma, un agente $\tau.P$ evolucionará tarde o temprano a P.

- Un agente $\bar{x}\tilde{y}.P$ (*output-prefixed agent*) envía los nombres de enlaces $\tilde{y}$ (*objects*) junto con el nombre x (*subject*) y después se comporta como P.

- Un agente $x(\tilde{w}).P$ (*input-prefixed agent*) espera hasta que reciba un conjunto de nombres $\tilde{y}$ precedido por el nombre x, y después se comporta como $P\{\tilde{y}/\tilde{w}\}$, en donde $\{\tilde{y}/\tilde{w}\}$ indica la substitución de $\tilde{w}$ por $\tilde{y}$.

- El operador de composición paralela trabaja en la forma esperada: $P \mid Q$ representa a P y Q actuando en paralelo.

- El operador suma se utiliza para expresar alternativas: $P + Q$ puede proceder o bien como P o como Q. La elección puede tomarse tanto de forma local como global. Utilizando la elección global dos agentes pueden sincronizarse mediante acciones complementarias, lo que da lugar a la regla principal de comunicación en el π-calculus:

  $$(\cdots + \bar{x}\tilde{y}.P + \cdots) \mid (\cdots + x(\tilde{w}).Q + \cdots) \xrightarrow{\tau} P \mid Q\{\tilde{y}/\tilde{w}\}$$

  Por otro lado, la elección local se expresan combinando el operador suma con las acciones silenciosas de τ. De esta forma, un agente $(\cdots + \tau.P + \tau.Q + \cdots)$ puede proceder como P o como Q independientemente de su contexto.

- Finalmente, $A(\tilde{w})$ se usa para definir nuevos agentes. Para ello se utilizan ecuaciones del tipo $A(\tilde{w}) = P$, que pueden ser recursivas.

El π-cálculo también se utiliza para razonar sobre los tipos de procesos y definir relaciones de equivalencia entre ellos, como por ejemplo las de bisimilitud débil y fuerte. Asimismo, la elevada expresividad de este tipo de formalismos para modelar los aspectos dinámicos de los sistemas ha servido para que numerosos autores los escojan como notaciones formales en sus trabajos. De esta forma se han definido varios lenguajes de descripción de arquitecturas software basados en CSP y π-cálculo. El uso de estas notaciones permite un tratamiento formal de las propiedades de los sistemas especificados con ellas, como puede ser la ausencia de bloqueos, la inferencia de resultados, o el estudio de la compatibilidad y reemplazabilidad de sus distintas partes.

Ahora bien, este tipo de formalismos son muy expresivos para tratar los aspectos dinámicos de los sistemas, pero sin embargo presentan algunas carencias al tratar sus aspectos estáticos. De ahí que normalmente se utilicen para representar las comunicaciones entre los componentes de las arquitecturas (la topología del sistema), pero no sus entidades.

7 Otros formalismos y combinaciones entre ellos

Además de los formalismos descritos anteriormente, también destacaremos aquí otras notaciones de uso extendido entre la comunidad software que trabaja en la aplicación de métodos formales para la construcción de aplicaciones en sistemas abiertos y distribuidos.

Las máquinas de estado comunicantes Son notaciones que permiten describir diferentes máquinas de estado que se comunican mediante paso de mensajes. Usualmente también incluyen mecanismos para crear y destruir procesos, definir estructuras de datos, e incluir secuencias de control al estilo de los lenguajes imperativos. Los lenguajes más conocidos que siguen este tipo de notaciones son SDL [ITU-T, 1994], ESTELLE [ISO, 1989a] y PROMELA [Holzmann, 1991].

Redes de Petri Este tipo de notación está también basada en las transiciones de estados para representar a los sistemas, aunque modela todo el sistema como un bloque en vez de representar los componentes como procesos. Las transiciones del sistema obedecen a un modelo de concurrencia real y muy apropiado para describir aplicaciones de tiempo real, por ejemplo.

La máquina química abstracta Conocido como CHAM por sus iniciales en inglés (*Chemical Abstract Machine*) [Berry y Boudol, 1992], este modelo ofrece una semántica puramente concurrente, pues representa la interacción y sincronización entre los procesos obedeciendo el modelo que siguen las moléculas químicas para reaccionar entre sí, y de ahí su nombre. Los elementos fundamentales en este modelo son las *moléculas*, las *soluciones* y las *reglas*. Una máquina es un triplete (G, C, R), donde G es una gramática, C es un conjunto de configuraciones (es decir, el lenguaje generado por la gramática), también llamadas moléculas, y R es un conjunto de reglas de la forma $condicion(C) \times \text{bag } C \times \text{bag } C$. Una solución es un multiconjunto de moléculas (bag C). Las reglas pueden dispararse en paralelo si no afectan a las mismas moléculas, y de aquí la concurrencia del modelo. En caso de conflictos entre reglas por afectar a moléculas comunes, se dispara sólo una de las reglas, escogida de forma no determinista.

B AMN Como parte del modelo B desarrollado en [Abrial, 1986], la notación de máquina abstracta de B (B AMN) es una notación formal orientada a objetos basada en la lógica de la pre-condición más débil de Dijkstra (*wp*) que ha sido utilizada con éxito en desarrollos industriales, sobre todo en Inglaterra (al igual que ocurre con Z). Actualmente se está intentando utilizar también para la especificación de sistemas reactivos y distribuidos [Butler, 1997] [Lano, 1997], exténdiéndola o combinándola con otros formalismos para aliviar sus puntos más débiles (no dispone de soporte semántico para la concurrencia, como le ocurre también a Z)

Además de en estos modelos, también se está trabajando en el uso combinado de ellos. La idea es que cada uno permite expresar mejor unos aspectos que otros, pero no hay ninguno que consiga expresar de forma satisfactoria todos los aspectos que intervienen en la especificación de una aplicación en un sistema abierto y distribuido. Por ejemplo, algunas notaciones son muy expresivas a la hora de describir las estructuras estáticas de un sistema (como puede ser Z) pero sin embargo no describen bien algunos aspectos dinámicos como la concurrencia y el tiempo real. Otros formalismos, más orientados a la movilidad, son capaces de expresar estos aspectos dinámicos (como el π-cálculo), y en particular se muestran muy apropiados para modelar las relaciones entre los componentes; sin embargo, se muestran pobres a la hora de representar la estructura interna y comportamiento de dichos componentes. Por esta razón varios autores combina formalismos. Entre ellos citaremos los siguientes:

- [Benjamin, 1989] combina Z y CSP para especificar un sistema basado en paso de mensajes. CSP especifica el comportamiento dinámico del sistema, mientras que la arquitectura del sistema y las estructuras de datos se especifican en Z. El problema de este ejemplo es que la combinación de ambas notaciones no se formaliza adecuadamente.

- [He, 1995] combina Z con las Redes de Petri para especificar el control de flujo, las relaciones causales y el comportamiento dinámico de sistemas cuyos aspectos estáticos hayan sido especificados en Z. En este caso tampoco se formaliza la combinación entre ambas notaciones.

- [Evans, 1994b] utiliza también Z y Redes de Petri, pero la integración es mejor que la del modelo anterior, pues Evans 'traduce' las Redes de Petri a Z, y utiliza la representación que ofrecen las Redes de Petri para visualizar sistemas especificados en Z.

- El mismo autor utiliza ese mismo año una lógica similar a la de Unity para dotar a Z de los aspectos de concurrencia que mencionábamos antes [Evans, 1994a].
- En [Ciancarini y Mascolo, 1997] se utiliza el modelo semántico de CHAM para dotar también a Z de esos aspectos de concurrencia, apoyándose en una lógica similar a la de Unity para razonar sobre los sistemas, que consigue unificar de forma muy natural tanto con el modelo de CHAM como con Z.
- Por último, [Mahony y Dong, 1998] combinan Object-Z y una extensión de CSP con tiempo (Timed-CSP) para construir TCOZ, una notación formal que une ambos formalismos para poder especificar, de manera natural, sistemas complejos cuyos componentes disponen de sus propias hebras de control.

Antes de acabar, nos gustaría hacer referencia a tres series de conferencias específicas sobre métodos formales con especial énfasis en la especificación de sistemas abiertos y distribuidos: la primera es FMOODS (Formal Methods for Object-Oriented Distributed Systems), una serie de conferencias organizadas por IFIP para el desarrollo y aplicación de los métodos formales a los sistemas abiertos y distribuidos, que se organiza cada 18 meses. En segundo lugar está la reunión anual de usuarios de Z (ZUM), con un capítulo dedicado a los sistemas reactivos. Y por último FME (Formal Methods Europe), ahora integrada con FM (a nivel mundial) que es la reunión anual europea de métodos formales, con especial énfasis en aplicaciones industriales de las notaciones formales.

Referencias

[Abrial, 1986] Abrial, J. (1986). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

[Alpern y Schneider, 1985] Alpern, B. y Schneider, F. (1985). Defining Liveness. *Information Processing Letters*, 21(4):181–185.

[Apt et al., 1980] Apt, K., Francez, N., y de Roever, W. (1980). A Proof System for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385.

[Ashcroft, 1975] Ashcroft, E. (1975). Proving Assertions About Parallel Programs. *Journal of Computer and System Sciences*, 10:110–135.

[Benjamin, 1989] Benjamin, M. (1989). A Message Passing System. An Example of Combining Z and CSP. En Nicholls, J. (ed.), *Proc. of the 4th Z Users Meeting (ZUM'89)*, Workshops in Computing, pp. 221–228. Springer-Verlag.

[Berry y Boudol, 1992] Berry, G. y Boudol, G. (1992). The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248.

[Brien, 1995] Brien, S. (1995). *A Model and Logic for Generically Typed Set Theory (Z)*. Tesis Doctoral, University of Oxford.

[Butler, 1997] Butler, M. (1997). An Approach to the Design of Distributed Systems with B AMN. En Bowen, J. P., Hinchey, M. G., y Till, D. (eds.), *Proc. of ZUM'97: The Z Formal Specification Notation*, núm. 1212 de LNCS, pp. 223–241. Springer-Verlag.

[Chandy y Kesselman, 1992] Chandy, K. M. y Kesselman, C. (1992). The Derivation of Compositional Programs. En *Proc. of the 1992 International Joint Conference and Symposium on Logic Programming*. MIT Press.

[Chandy y Misra, 1988] Chandy, K. M. y Misra, J. (1988). *Parallel Program Design*. Addison-Wesley.

[Ciancarini y Mascolo, 1997] Ciancarini, P. y Mascolo, C. (1997). Analyzing and Refining an Architectural Style. En Bowen, J. P., Hinchey, M. G., y Till, D. (eds.), *Proc. of ZUM'97: The Z Formal Specification Notation*, núm. 1212 de LNCS, pp. 349–368. Springer-Verlag.

[Clavel et al.,] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., y Quesada, J. Maude: Specification and Programming in Rewriting Logic. Manuscript, SRI International, 1999. Available at `http://maude.csl.sri.com`.

[Duke et al., 1996] Duke, R., Bailes, C., y Smith, G. (1996). A Blocking Model for Reactive Objects. *Formal Aspects of Computing*, 8(3):347–368.

[Duke et al., 1994] Duke, R., Rose, G., y Smith, G. (1994). Object-Z: A Specification Language Advocated for the Description of Standards. Informe Técnico núm. 94–45, University of Queensland.

[Engberg y Nielsen, 1986] Engberg, U. y Nielsen, M. (1986). A Calculus of Communicating Systems with Label-passing. Informe Técnico núm. DAIMI PB-208, Computer Science Dept., University of Aarhus.

[Evans, 1994a] Evans, A. S. (1994a). Specifying and Verifying Concurrent Systems Using Z. En *Proc. of FME'94: Industrial Benefits of Formal Methods*, núm. 873 de LNCS, Barcelona.

[Evans, 1994b] Evans, A. S. (1994b). Visualizing Concurrent Z Specifications. En Bowen, J. y Hall, J. (eds.), *Proc. of the 8th Z Users Meeting (ZUM'94)*, Workshops in Computing, pp. 269–281. Springer-Verlag.

[Evans, 1997] Evans, A. S. (1997). An Improved recipe for Specifying Reactive Systems in Z. En Bowen, J. P., Hinchey, M. G., y Till, D. (eds.), *Proc. of ZUM'97: The Z Formal Specification Notation*, núm. 1212 de LNCS, pp. 275–294. Springer-Verlag.

[Floyd, 1967] Floyd, R. W. (1967). Assigning Meaning to Programs. En *Proc. of the Symposium on Applied Maths*, volumen 19, pp. 19–32. AMS.

[Gerth y de Roever, 1984] Gerth, R. y de Roever, W. (1984). A Proof System for Concurrent Ada Programs. *Science of Computer Programming*, 4(2):159–204.

[Guttag y Horning, 1993] Guttag, J. V. y Horning, J. J. (1993). *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag.

[Hall y Martin, 1997] Hall, J. y Martin, A. (1997). $\mathcal{W}$ Reconstructed. En Bowen, J. P., Hinchey, M. G., y Till, D. (eds.), *Proc. of ZUM'97: The Z Formal Specification Notation*, núm. 1212 de LNCS, pp. 115–134. Springer-Verlag.

[He, 1995] He, X. (1995). PZ Nets: A Formal Method Integrating Petri Nets and Z. En *Proc. of the 7th International Conference on Software Engineering and Knowledge Engineering*, pp. 173–180, Rockville, Maryland.

[Hoare, 1967] Hoare, C. (1967). An Axiomatic Basis for Computer Programming. *Communications of the Association for Computing Machinery*, 12(10):576–583.

[Hoare, 1984] Hoare, C. (1984). *Communicating Sequential Processes*. Prentice-Hall.

[Holzmann, 1991] Holzmann, G. (1991). *Design and Validation of Computer Protocols*. Prentice-Hall.

[ISO, 1989a] ISO (1989a). Information Processing Systems – Open Systems Interconnection. ESTELLE, A Formal Description Technique Based on an Extended State Transition Model. Rec. ISO 9074, ISO.

[ISO, 1989b] ISO (1989b). Information Processing Systems – Open Systems Interconnection. LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior. Rec. ISO 8807, ISO.

[ITU-T, 1994] ITU-T (1994). SDL: Specification and Description Language. Rec. Z.100, ITU-T.

[Jones, 1994] Jones, C. B. (1994). *Systematic Software Development using VDM*. Prentice-Hall, 2 edición.

[Lamport, 1991] Lamport, L. (1991). The Temporal Logic of Actions. Informe Técnico núm. 79, DEC Corporation, Systems Research Centre.

[Lamport, 1993] Lamport, L. (1993). Verification and Specification of Concurrent Programs. En de Bakker, J., de Roeveer, W., y Rozenberg, G. (eds.), *A Decade of Concurrency*, núm. 803 de LNCS, pp. 347–374. Springer-Verlag.

[Lamport, 1994] Lamport, L. (1994). TLZ. En *Workshops in Computing*, pp. 267–268. Springer-Verlag.

[Lano, 1997] Lano, K. (1997). Specifying Reactive Systems in B AMN. En Bowen, J. P., Hinchey, M. G., y Till, D. (eds.), *Proc. of ZUM'97: The Z Formal Specification Notation*, núm. 1212 de LNCS, pp. 242–274. Springer-Verlag.

[Levin y Gries, 1981] Levin, G. y Gries, D. (1981). An Proof Technique for Communicating Sequential Programs. *Acta Informatica*, 15(3):281–302.

[Mahony y Dong, 1998] Mahony, B. y Dong, J. S. (1998). Network Topology and a Case Study in TCOZ. En Bowen, J. P., Fett, A., y Hinchey, M. G. (eds.), *Proc. of ZUM'98: The Z Formal Specification Notation*, núm. 1493 de LNCS, pp. 308–327. Springer-Verlag.

[Meseguer, 1993] Meseguer, J. (1993). A Logical Theory of Concurrent Objects and its Realization in the Maude Language. En Agha, G., Wegner, P., y Yonezawa, A. (eds.), *Research Directions in Object-Based Concurrency*, pp. 314–390. The MIT Press.

[Meseguer, 2000] Meseguer, J. (2000). Rewriting Logic and Maude: A Wide-Spectrum Semantic Framework for Object-Based Distributed Systems. En Smith, S. F. y Talcott, C. L. (eds.), *Proc. of FMOODS'2000*, pp. 89–117, Stanford, CA. Kluwer Academic Publishers.

[Milner, 1989] Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

[Milner et al., 1992] Milner, R., Parrow, J., y Walker, D. (1992). A Calculus of Mobile Processes. *Journal of Information and Computation*, 100:1–77.

[Misra, 1990] Misra, J. (1990). Specifying Concurrent Objects as Communicating Processes. *Science of Computer Programming*, 14(2–3):159–184.

[Moszkowski, 1986] Moszkowski, B. (1986). *Executing Temporal Logic Programs*. Cambridge University Press.

[Owicki y Gries, 1976] Owicki, S. y Gries, D. (1976). An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*, 6(4):319–340.

[Owicki y Lamport, 1982] Owicki, S. y Lamport, L. (1982). Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495.

[Pnueli, 1977] Pnueli, A. (1977). The Temporal Logic of Programs. En *Proc. of the 18th Annual Symposium on the Foundations of Computer Science*, pp. 46–57. IEEE.

[Smith, 1994] Smith, G. (1994). A Logic for Object-Z. Informe Técnico núm. 94–48, University of Queensland.

[Spivey, 1992] Spivey, J. (1992). *The Z Notation. A Reference Manual. 2nd Ed.* Prentice Hall.

[Stepney et al., 1992] Stepney, S., Barden, R., y Cooper, D. (eds.) (1992). *Object Orientation in Z*. Workshops in Computing. Springer-Verlag.

[Stepney et al., 1998] Stepney, S., Cooper, D., y Woodcock, J. (1998). More Powerful Z Data Refinement: Pushing the State of the Art in Industrial Refinement. En Bowen, J. P., Fett, A., y Hinchey, M. G. (eds.), *Proc. of ZUM'98: The Z Formal Specification Notation*, núm. 1493 de LNCS, pp. 284–307. Springer-Verlag.

[Woodcock y Brien, 1992] Woodcock, J. y Brien, S. (1992). $\mathcal{W}$: A Logic for Z. En Nicholls, J. (ed.), *Proc. of the 6th Z User Meeting*. Springer-Verlag.

[Woodcock y Davies, 1996] Woodcock, J. y Davies, J. (1996). *Using Z: Specification, Refinement and Proof*. Prentice Hall.