Lenguajes de Descripción de Arquitectura (ADL)

Versión 1.0 – Marzo de 2004

Carlos Reynoso – Nicolás Kicillof
UNIVERSIDAD DE BUENOS AIRES

Introducción

Una vez que el arquitecto de software, tras conocer el requerimiento, se decide a delinear su estrategia y a articular los patrones que se le ofrecen hoy en profusión, se supone que debería expresar las características de su sistema, o en otras palabras, modelarlo, aplicando una convención gráfica o algún lenguaje avanzado de alto nivel de abstracción.

A esta altura del desarrollo de la arquitectura de software, podría pensarse que hay abundancia de herramientas de modelado que facilitan la especificación de desarrollos basados en principios arquitectónicos, que dichas herramientas han sido consensuadas y estandarizadas hace tiempo y que son de propósito general, adaptables a soluciones de cualquier mercado vertical y a cualquier estilo arquitectónico. La creencia generalizada sostendría que modelar arquitectónicamente un sistema se asemeja al trabajo de articular un modelo en ambientes ricos en prestaciones gráficas, como es el caso del modelado de tipo CASE o UML, y que el arquitecto puede analizar visualmente el sistema sin sufrir el aprendizaje de una sintaxis especializada. También podría pensarse que los instrumentos incluyen la posibilidad de diseñar modelos correspondientes a proyectos basados en tecnología de internet, Web services o soluciones de integración de plataformas heterogéneas, y que, una vez trazado el modelo, el siguiente paso en el ciclo de vida de la solución se produzca con naturalidad y esté servido por ténicas bien definidas. Como se verá en este documento, la situación es otra, dista de ser clara y es harto más compleja.

En primer lugar, el escenario de los web services ha forzado la definición de un estilo de arquitectura que no estaba contemplado a la escala debida en el inventario canónico de tuberías y filtros, repositorio, eventos, capas, llamada y retorno/OOP y máquinas virtuales. El contexto de situación, como lo reveló la impactante tesis de Roy Fielding sobre REST [Fie00], es hoy en día bastante distinto al de los años de surgimiento de los estilos y los lenguajes de descripción de arquitecturas (en adelante, ADLs). Los ADLs se utilizan, además, para satisfacer requerimientos descriptivos de alto nivel de abstracción que las herramientas basadas en objeto en general y UML en particular no cumplen satisfactoriamente. Entre las comunidades consagradas al modelado OO y la que patrocina o frecuenta los ADLs (así como entre las que se inclinan por el concepto de estilos arquitectónicos y las que se trabajan en función de patrones) existen relaciones complejas que algunas veces son de complementariedad y otras de antagonismo.

Aunque algunos arquitectos influyentes, como Jørgen Thelin, alegan que el período de gloria de la OOP podría estar acercándose a su fin, el hecho concreto es que el modelado orientado a objetos de sistemas basados en componentes posee ciertos número de rasgos muy convenientes a la hora de diseñar o al menos describir un sistema. En primer lugar, las notaciones de objeto resultan familiares a un gran número de ingenieros de software.

Proporcionan un mapeo directo entre un modelo y una implementación y se dispone de un repertorio nutrido de herramientas comerciales para trabajar con ellas; implementan además métodos bien definidos para desarrollar sistemas a partir de un conjunto de requerimientos.

Pero la descripción de sistemas basados en componentes presenta también limitaciones serias, que no son de detalle sino más bien estructurales. En primer lugar, sólo proporcionan una única forma de interconexión primitiva: la invocación de método. Esto hace difícil modelar formas de interacción más ricas o diferentes. Ya en los días del OMT, Rumbaugh y otros admitían cándidamente que "el *target* de implementación más natural para un diseño orientado a objetos es un lenguaje orientado a objetos" [RBP+91:296]. En segundo orden, el soporte de los modelos OO para las descripciones jerárquicas es, en el mejor de los casos, débil. Tampoco se soporta la definición de familias de sistemas o estilos; no hay recurso sintáctico alguno para caracterizar clases de sistemas en términos de las restricciones de diseño que debería observar cada miembro de la familia. En última instancia, no brindan soporte formal para caracterizar y analizar propiedades no funcionales, lo que hace difícil razonar sobre propiedades críticas del sistema, tales como performance y robustez [GMW00]. A la luz de estas observaciones y de otras que saldrán a la luz en el capítulo sobre UML de este mismo documento, está claro cuál es el nicho vacante que los ADLs vinieron a ocupar.

Según señala Mary Shaw [Shaw94] y Neno Medvidovic [Med96] en sus revisiones de los lenguajes de descripción arquitectónica, el uso que se da en la práctica y en el discurso a conceptos tales como arquitectura de software o estilos suele ser informal y ad hoc. En la práctica industrial, las configuraciones arquitectónicas se suelen describir por medio de diagramas de cajas y líneas, con algunos añadidos diacríticos; los entornos para esos diagramas suelen ser de espléndida calidad gráfica, comunican con relativa efectividad la estructura del sistema y siempre brindan algún control de consistencia respecto de qué clase de elemento se puede conectar con cuál otra; pero a la larga proporcionan escasa información sobre la computación efectiva representada por las cajas, las interfaces expuestas por los componentes o la naturaleza de sus computaciones.

En la década de 1990 y en lo que va del siglo XXI, sin embargo, se han materializado diversas propuestas para describir y razonar en términos de arquitectura de software; muchas de ellas han asumido la forma de ADLs. Estos suministran construcciones para especificar abstracciones arquitectónicas y mecanismos para descomponer un sistema en componentes y conectores, especificando de qué manera estos elementos se combinan para formar configuraciones y definiendo familias de arquitecturas o estilos. Contando con un ADL, un arquitecto puede razonar sobre las propiedades del sistema con precisión, pero a un nivel de abstracción convenientemente genérico. Algunas de esas propiedades podrían ser, por ejemplo, protocolos de interacción, anchos de banda y latencia, localización del almacenamiento, conformidad con estándares arquitectónicos y previsiones de evolución ulterior del sistema.

Hasta la publicación de las sistematizaciones de Shaw y Garlan, Kogut y Clements o Nenan Medvidovic existía relativamente poco consenso respecto a qué es un ADL, cuáles lenguajes de modelado califican como tales y cuáles no, qué aspectos de la arquitectura se deben modelar con un ADL y cuáles de éstos son más adecuados para modelar qué

clase de arquitectura o estilo. También existía (y aún subsiste) cierta ambigüedad a propósito de la diferencia entre ADLs, especificaciones formales (como CHAM, cálculo Л o Z), lenguajes de interconexión de módulos (MIL) como MIL75 o Intercol, lenguajes de modelado de diseño (UML), herramientas de CASE y hasta determinados lenguajes con reconocidas capacidades modelizadoras, como es el caso de CODE, un lenguaje de programación paralela que en opinión de Paul Clements podría tipificar como un ADL satisfactorio.

La definición de ADL que habrá de aplicarse en lo sucesivo es la de un lenguaje descriptivo de modelado que se focaliza en la estructura de alto nivel de la aplicación antes que en los detalles de implementación de sus módulos concretos [Ves93]. Los ADL que existen actualmente en la industria rondan la cifra de veinticinco; cuatro o cinco de ellos han experimentado dificultades en su desarrollo o no se han impuesto en el mercado de las herramientas de arquitectura. Todos ellos oscilan entre constituirse como ambientes altamente intuitivos con eventuales interfaces gráficas o presentarse como sistemas rigurosamente formales con profusión de notación simbólica, en la que cada entidad responde a algún teorema. Alrededor de algunos ADLs se ha establecido una constelación de herramientas de análisis, verificadores de modelos, aplicaciones de generación de código, *parsers*, soportes de *runtime*, etcétera. El campo es complejo, y de la decisión que se tome puede depender el éxito o el fracaso de un proyecto.

No existe hasta hoy una definición consensuada y unívoca de ADL, pero comúnmente se acepta que un ADL debe proporcionar un modelo explícito de componentes, conectores y sus respectivas configuraciones. Se estima deseable, además, que un ADL suministre soporte de herramientas para el desarrollo de soluciones basadas en arquitectura y su posterior evolución.

La literatura referida a los ADL es ya de magnitud respetable. Aquí hemos tomado en consideración los estudios previos de Kogut y Clements [KC94, KC95, Cle96a], Vestal [Ves93], Luckham y Vera [LV95], Shaw, DeLine y otros [SDK+95], Shaw y Garlan [SG94, SG95] y Medvidovic [Med96], así como la documentación de cada uno de los lenguajes. Los principales ADLs que se revisarán en este estudio son ACME, Armani, Aesop, ArTek, C2, Darwin, Jacal, LILEANNA, MetaH/AADL, Rapide, UniCon, Weaves, Wright y xADL. Sobre unos cuantos no incluidos en esa lista no disponemos de demasiada información ni de experiencia concreta: Adage, ASDL, Avionics ADL, FR, Gestalt, Koala, Mae, PSDL/CAPS, QAD, SAM; serán por ende dejados al margen. A pesar de las afirmaciones de Clements [Cle95], no cabe considerar que UNAS/SALE, un producto comercial para gestión del ciclo de vida de un desarrollo, constituya un ADL. No habrá de tratarse aquí tampoco, en general, de ADLs que están en proceso de formulación, como algunos estándares emergentes o los Domain Specific Languages (DSL) que Microsoft se encuentra elaborando para su inclusión en la próxima encarnación de VisualStudio ("Whidbey"). Estos serán más bien lenguajes de patrones, a los que el ADL deberá tomar en cuenta como a un elemento de juicio adicional.

La delimitación categórica de los ADLs es problemática. Ocasionalmente, otras notaciones y formalismos se utilizan como si fueran ADLs para la descripción de arquitecturas. Casos a cuento serían CHAM, UML y Z. Aunque se hará alguna referencia a ellos, cabe puntualizar que no son ADLs en sentido estricto y que, a pesar de los

estereotipos históricos (sobre todo en el caso de UML), no se prestan a las mismas tareas que los lenguajes descriptivos de arquitectura, diseñados específicamente para esa finalidad. Aparte de CODE, que es claramente un lenguaje de programación, se excluirá aquí entonces toda referencia a Modechart y PSDL (lenguajes de especificación o prototipado de sistemas de tiempo real), LOTOS (un lenguaje de especificación formal, asociado a entornos de diseño como CADP), ROOM (un lenguaje de modelado de objetos de tiempo real), Demeter (una estrategia de diseño y programación orientada a objetos), Resolve (una técnica matemáticamente fundada para desarrollo de componentes re-utilizables), Larch y Z (lenguajes formales de especificación) por más que todos ellos hayan sido asimilados a ADLs por algunos autores en más de una ocasión. Se hará una excepción con UML, sin embargo. A pesar de no calificar en absoluto como ADL, se ha probado que UML puede utilizarse no tanto como un ADL por derecho propio, sino como metalenguaje para simular otros ADLs, y en particular C2 y Wright [RMR98]. Otra excepción concierne a CHAM, por razones que se verán más adelante.

ADL	Fecha	Investigador - Organismo	Observaciones
Acme	1995	Monroe & Garlan (CMU), Wile (USC)	Lenguaje de intercambio de ADLs
Aesop	1994	Garlan (CMU)	ADL de propósito general, énfasis en estilos
ArTek	1994	Terry, Hayes-Roth, Erman (Teknowledge, DSSA)	Lenguaje específico de dominio - No es ADL
Armani	1998	Monroe (CMU)	ADL asociado a Acme
C2 SADL	1996	Taylor/Medvidovic (UCI)	ADL específico de estilo
CHAM	1990	Berry / Boudol	Lenguaje de especificación
Darwin	1991	Magee, Dulay, Eisenbach, Kramer	ADL con énfasis en dinámica
Jacal	1997	Kicillof , Yankelevich (Universidad de Buenos Aires)	ADL - Notación de alto nivel para descripción y prototipado
LILEANNA	1993	Tracz (Loral Federal)	Lenguaje de conexión de módulos
MetaH	1993	Binns, Englehart (Honeywell)	ADL específico de dominio
Rapide	1990	Luckham (Stanford)	ADL & simulación
SADL	1995	Moriconi, Riemenschneider (SRI)	ADL con énfasis en mapeo de refinamiento
UML	1995	Rumbaugh, Jacobson, Booch (Rational)	Lenguaje genérico de modelado – No es ADL
UniCon	1995	Shaw (CMU)	ADL de propósito general, énfasis en conectores y estilos
Wright	1994	Garlan (CMU)	ADL de propósito general, énfasis en comunicación
xADL	2000	Medvidovic, Taylor (UCI, UCLA)	ADL basado en XML

El objetivo del presente estudio es dar cuenta del estado de arte en el desarrollo de los ADLs en el momento actual (2004), varios años después de realizados los *surveys* más reputados, los cuales son a su vez anteriores a la explosión de los web services, SOAP, el modelo REST y las arquitecturas basadas en servicios. También se examinará detalladamente la disponibilidad de diversos ADLs y herramientas concomitantes en ambientes Windows, la forma en que los ADLs engranan con la estrategia general de Microsoft en materia de arquitectura y su relación con el campo de los patrones arquitectónicos.

En lugar de proceder a un análisis de rasgos, examinando la forma en que los distintos ADLs satisfacen determinados requerimientos, aquí se ha optado por describir cada ADL

en función de variables que a veces son las mismas en todos los casos y en otras oportunidades son específicas de la estructura de un ADL en particular. Nos ha parecido de interés subrayar el modelo computacional en que los ADLs se basan, a fin de matizar la presentación de un campo que hasta hace poco se estimaba coto exclusivo de los paradigmas orientados a objeto. Mientras éstos se están elaborando mayormente en organismos de estandarización, casi todos los ADLs se originan en ámbitos universitarios. Crucial en el tratamiento que aquí se dará a los ADLs es la capacidad de trabajar en términos de estilos, que se estudiarán en un documento separado [Rey04]. Un estilo define un conjunto de propiedades compartidas por las configuraciones que son miembros de él. Estas propiedades pueden incluir un vocabulario común y restricciones en la forma en que ese vocabulario puede ser utilizado en dichas configuraciones. Tengan a no los distintos ADLs soportes de estilo, patrones o lo que fuese, no interesa aquí tanto describir o asignar puntaje a productos o paquetes que por otro lado son cambiantes, sino caracterizar problemas, ideas, paradigmas y escenarios de modelado.

Criterios de definición de un ADL

Los ADLs se remontan a los lenguajes de interconexión de módulos (MIL) de la década de 1970, pero se han comenzado a desarrollar con su denominación actual a partir de 1992 o 1993, poco después de fundada la propia arquitectura de software como especialidad profesional. La definición más simple es la de Tracz [Wolf97] que define un ADL como una entidad consistente en cuatro “Cs”: componentes, conectores, configuraciones y restricciones (*constraints*). Una de las definiciones más tempranas es la de Vestal [Ves93] quien sostiene que un ADL debe modelar o soportar los siguientes conceptos:

- Componentes
- Conexiones
- Composición jerárquica, en la que un componente puede contener una sub-arquitectura completa
- Paradigmas de computación, es decir, semánticas, restricciones y propiedades no funcionales
- Paradigmas de comunicación
- Modelos formales subyacentes
- Soporte de herramientas para modelado, análisis, evaluación y verificación
- Composición automática de código aplicativo

Basándose en su experiencia sobre Rapide, Luckham y Vera [LV95] establecen como requerimientos:

- Abstracción de componentes
- Abstracción de comunicación
- Integridad de comunicación (sólo los componentes que están conectados pueden comunicarse)
- Capacidad de modelar arquitecturas dinámicas
- Composición jerárquica

- Relatividad (o sea, la capacidad de mapear o relacionar conductas entre arquitecturas)

Tomando como parámetro de referencia a UniCon, Shaw y otros [SDK+95] alegan que un ADL debe exhibir:

- Capacidad para modelar componentes con aserciones de propiedades, interfaces e implementaciones
- Capacidad de modelar conectores con protocolos, aserción de propiedades e implementaciones
- Abstracción y encapsulamiento
- Tipos y verificación de tipos
- Capacidad para integrar herramientas de análisis

Otros autores, como Shaw y Garlan [SG94] estipulan que en los ADLs los conectores sean tratados explícitamente como entidades de primera clase (lo que dejaría al margen de la lista a dos de ellos al menos) y han afirmado que un ADL genuino tiene que proporcionar propiedades de composición, abstracción, reusabilidad, configuración, heterogeneidad y análisis, lo que excluiría a todos los lenguajes convencionales de programación y a los MIL.

La especificación más completa y sutil (en tanto que se presenta como provisional, lo que es propio de un campo que recién se está definiendo) es la de Medvidovic [Med96]:

- Componentes
  Interfaz
  Tipos
  Semántica
  Restricciones (*constraints*)
  Evolución
  Propiedades no funcionales
- Conectores
  Interfaz
  Tipos
  Semántica
  Restricciones
  Evolución
  Propiedades no funcionales
- Configuraciones arquitectónicas
  Comprensibilidad
  Composicionalidad
  Heterogeneidad
  Restricciones
  Refinamiento y trazabilidad
  Escalabilidad
  Evolución
  Dinamismo

Propiedades no funcionales
- Soporte de herramientas
Especificación activa
Múltiples vistas
Análisis
Refinamiento
Generación de código
Dinamismo

Nótese, en todo caso, que no se trata tanto de aspectos definitorios del concepto de ADL, sino de criterios para la evaluación de los ADLs existentes, o sea de un marco de referencia para la clasificación y comparación de los ADLs.

En base a las propuestas señaladas, definiremos a continuación los elementos constitutivos primarios que, más allá de la diversidad existente, son comunes a la ontología de todos los ADLs y habrán de ser orientadores de su tratamiento en este estudio.

- Componentes: Representan los elementos computacionales primarios de un sistema. Intuitivamente, corresponden a las cajas de las descripciones de caja-y-línea de las arquitecturas de software. Ejemplos típicos serían clientes, servidores, filtros, objetos, pizarras y bases de datos. En la mayoría de los ADLs los componentes pueden exponer varias interfaces, las cuales definen puntos de interacción entre un componente y su entorno.
- Conectores. Representan interacciones entre componentes. Corresponden a las líneas de las descripciones de caja-y-línea. Ejemplos típicos podrían ser tuberías (*pipes*), llamadas a procedimientos, *broadcast* de eventos, protocolos cliente-servidor, o conexiones entre una aplicación y un servidor de base de datos. Los conectores también tienen una especie de interfaz que define los roles entre los componentes participantes en la interacción.
- Configuraciones o sistemas. Se constituyen como grafos de componentes y conectores. En los ADLs más avanzados la topología del sistema se define independientemente de los componentes y conectores que lo conforman. Los sistemas también pueden ser jerárquicos: componentes y conectores pueden subsumir la representación de lo que en realidad son complejos subsistemas.
- Propiedades. Representan información semántica sobre un sistema más allá de su estructura. Distintos ADLs ponen énfasis en diferentes clases de propiedades, pero todos tienen alguna forma de definir propiedades no funcionales, o pueden admitir herramientas complementarias para analizarlas y determinar, por ejemplo, el *throughput* y la latencia probables, o cuestiones de seguridad, escalabilidad, dependencia de bibliotecas o servicios específicos, configuraciones mínimas de *hardware* y tolerancia a fallas.
- Restricciones. Representan condiciones de diseño que deben acatarse incluso en el caso que el sistema evolucione en el tiempo. Restricciones típicas serían restricciones en los valores posibles de propiedades o en las configuraciones topológicas

admisibles. Por ejemplo, el número de clientes que se puede conectar simultáneamente a un servicio.

- Estilos. Representan familias de sistemas, un vocabulario de tipos de elementos de diseño y de reglas para componerlos. Ejemplos clásicos serían las arquitecturas de flujo de datos basados en grafos de tuberías (*pipes*) y filtros, las arquitecturas de pizarras basadas en un espacio de datos compartido, o los sistemas en capas. Algunos estilos prescriben un *framework*, un estándar de integración de componentes, patrones arquitectónicos o como se lo quiera llamar.
- Evolución. Los ADLs deberían soportar procesos de evolución permitiendo derivar subtipos a partir de los componentes e implementando refinamiento de sus rasgos. Sólo unos pocos lo hacen efectivamente, dependiendo para ello de lenguajes que ya no son los de diseño arquitectónico sino los de programación.
- Propiedades no funcionales. La especificación de estas propiedades es necesaria para simular la conducta de *runtime*, analizar la conducta de los componentes, imponer restricciones, mapear implementaciones sobre procesadores determinados, etcétera.

Lenguajes

En la sección siguiente del documento, revisaremos algunos de los ADLs fundamentales de la arquitectura de software contemporánea en función de los elementos comunes de su ontología y analizando además su disponibilidad para la plataforma Windows, las herramientas gráficas concomitantes y su capacidad para generar código ejecutable, entre otras variables de relevancia.

Acme - Armani

Acme se define como una herramienta capaz de soportar el mapeo de especificaciones arquitectónicas entre diferentes ADLs, o en otras palabras, como un lenguaje de intercambio de arquitectura. No es entonces un ADL en sentido estricto, aunque la literatura de referencia acostumbra tratarlo como tal. De hecho, posee numerosas prestaciones que también son propias de los ADLs. En su sitio oficial se reconoce que como ADL no es necesariamente apto para cualquier clase de sistemas, al mismo tiempo que se destaca su capacidad de describir con facilidad sistemas “relativamente simples”.

El proyecto Acme comenzó a principios de 1995 en la Escuela de Ciencias de la Computación de la Universidad Carnegie Mellon. Hoy este proyecto se organiza en dos grandes grupos, que son el lenguaje Acme propiamente dicho y el Acme Tool Developer’s Library (AcmeLib). De Acme se deriva, en gran parte, el ulterior estándar emergente ADML. Fundamental en el desarrollo de Acme ha sido el trabajado de destacados arquitectos y sistematizadores del campo, entre los cuales el más conocido es sin duda David Garlan, uno de los teóricos de arquitectura de software más activos en la década de 1990. La bibliografía relevante para profundizar en Acme es el reporte de R. T. Monroe [Mon98] y el artículo de Garlan, Monroe y Wile [GMW00].

Objetivo principal – La motivación fundamental de Acme es el intercambio entre arquitecturas e integración de ADLs. Garlan considera que Acme es un lenguaje de

descripción arquitectónica de segunda generación; podría decirse que es de segundo orden: un metalenguaje, una *lingua franca* para el entendimiento de dos o más ADLs, incluido Acme mismo. Con el tiempo, sin embargo, la dimensión metalingüística de Acme fue perdiendo prioridad y los desarrollos actuales profundizan su capacidad intrínseca como ADL puro.

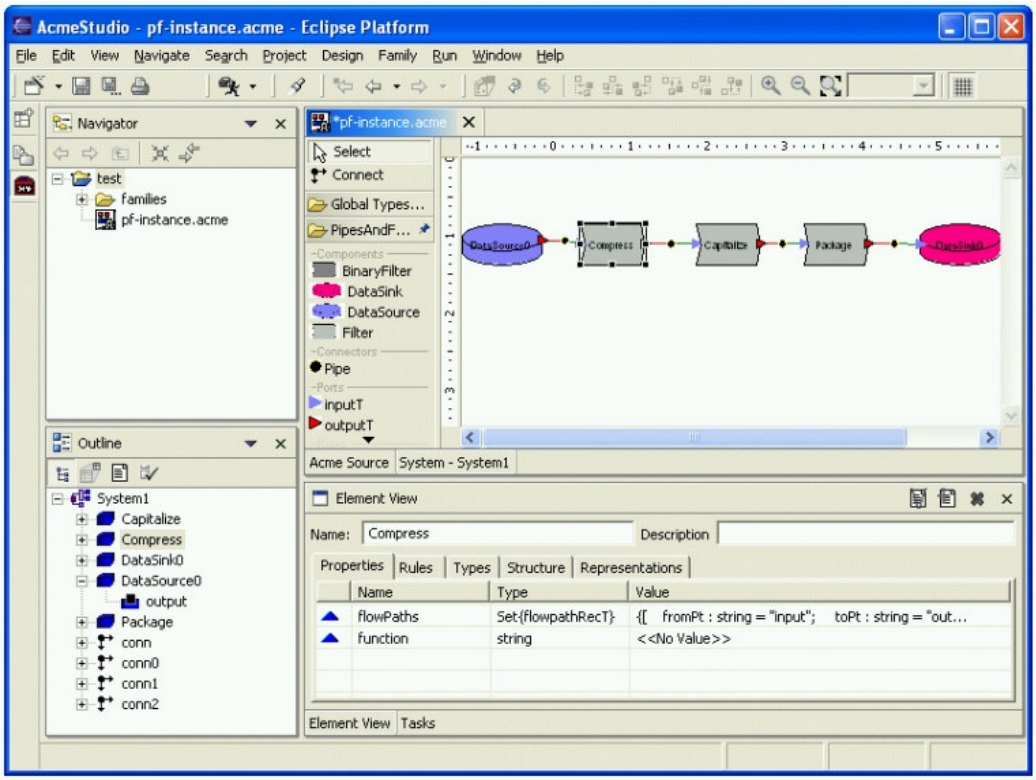


Fig. 1 - Ambiente de edición de AcmeStudio con diagrama de tubería y filtros

Sitio de referencia – El sitio de cabecera depende de la School of Computer Science de la Carnegie Mellon University en Pittsburgh. AcmeWeb se encuentra en http://www-2.cs.-cmu.edu/~acme/.

Acme soporta la definición de cuatro tipos de arquitectura: la estructura (organización de un sistema en sus partes constituyentes); las propiedades de interés (información que permite razonar sobre el comportamiento local o global, tanto funcional como no funcional); las restricciones (lineamientos sobre la posibilidad del cambio en el tiempo); los tipos y estilos. La estructura se define utilizando siete tipos de entidades: componentes, conectores, sistemas, puertos, roles, representaciones y rep-mapas (mapas de representación).

Componentes – Representan elementos computacionales y almacenamientos de un sistema. Como se verá en el ejemplo, un componente se define siempre dentro de una familia.

Interfaces – Todos los ADLs conocidos soportan la especificación de interfaces para sus componentes. En Acme cada componente puede tener múltiples interfaces. Igual que en Aesop y Wright los puntos de interfaz se llaman puertos (*ports*). Los puertos pueden definir interfaces tanto simples como complejas, desde una signatura de procedimiento hasta una colección de rutinas a ser invocadas en cierto orden, o un evento de multicast.

Conectores – En su ejemplar estudio de los ADLs existentes, Medvidovic (1996) llama a los lenguajes que modelan sus conectores como entidades de primera clase lenguajes de configuración explícitos, en oposición a los lenguajes de configuración *in-line*. Acme pertenece a la primera clase, igual que Wright y UniCon. Los conectores representan interacciones entre componentes. Los conectores también tienen interfaces que están definidas por un conjunto de roles. Los conectores binarios son los más sencillos: el invocador y el invocado de un conector RPC, la lectura y la escritura de un conector de tubería, el remitente y el receptor de un conector de paso de mensajes.

Semántica – Muchos lenguajes de tipo ADL no modelan la semántica de los componentes más allá de sus interfaces. En este sentido, Acme sólo soporta cierta clase de información semántica en listas de propiedades. Estas propiedades no se interpretan, y sólo existen a efectos de documentación.

Estilos – Acme posee manejo intensivo de familias o estilos. Esta capacidad está construida naturalmente como una jerarquía de propiedades correspondientes a tipos. Acme considera, en efecto, tres clase de tipos: tipos de propiedades, tipos estructurales y estilos. Así como los tipos estructurales representan conjuntos de elementos estructurales, una familia o estilo representa un conjunto de sistemas. Una familia Acme se define especificando tres elementos de juicio: un conjunto de tipos de propiedades y tipos estructurales, un conjunto de restricciones y una estructura por defecto, que prescribe el conjunto mínimo de instancias que debe aparecer en cualquier sistema de la familia. El uso del término "familia" con preferencia a "estilo" recupera una idea de uno de los precursores tempranos de la arquitectura de software, David Parnas [Par76].

El siguiente ejemplo define una familia o estilo de la clase más simple, tubería y filtros. Este estilo está relacionado con las arquitecturas de flujo de datos. Por ser el estilo más primitivo y de definición más escueta, se ha escogido ejemplificar con él la sintaxis de Acme y de todos los ADLs que se revisarán después. El estilo es característico no sólo de arquitecturas de tipo UNIX, sino que es ampliamente usado en compiladores, flujos de datos, tratamiento de XML con SAX, procesamiento de señales y programación funcional, así como en los mecanismos de procesamiento de consultas de algunos servidores de bases de datos. Es el único estilo, por otra parte, que puede ser implementado explícita o implícitamente en todos los ADLs [Cle96]. En Acme la sintaxis para definirlo sería:

```
// Una familia Acme incluye un conjunto de tipos de
// componente,conector, puerto (port) y rol que definen el vocabulario
// propio del estilo.
Family PipeFilterFam = {
  // Declara tipos de componente.
  // Una definición de tipo de componente en Acme permite establecer
  // la estructura requerida por el tipo. Esta estructura
  // se define mediante la misma sintaxis que la instancia
  // de un componente.
  Component Type FilterT = {
        // Todos los filtros definen por lo menos dos puertos
         Ports { stdin; stdout; };
        Property throughput : int;
  };

  // Extiende el tipo básico de filtro con una subclase (herencia)
  // Las instancia de WindowsFilterT tendrán todas las propiedades y
```

```
  // puertos de las instancias de FilterT, más un puerto stderr
  // y una propiedad implementationFile.

  Component Type WindowsFilterT extends FilterT with {
        Port stderr;
        Property implementationFile : String;
  };
  // Declara el tipo de conector de tubería. Igual que los
  // tipos de componente, un tipo de conector también describe
  // la estructura requerida.
  Connector Type PipeT = {
        Roles { source; sink; };
        Property bufferSize : int;
  };
   // Declara algunos tipos de propiedad que pueden usarse en
   // sistemas del estilo PipeFilterFam
   Property Type StringMsgFormatT = Record [ size:int; msg:String; ];
   Property Type TasksT = enum {sort, transform, split, merge};
};
```

La biblioteca AcmeLib define un conjunto de clases para manipular representaciones arquitectónicas Acme en cualquier aplicación. Su código se encuentra disponible tanto en C++ como en Java, y puede ser invocada por lo tanto desde cualquier lenguaje la plataforma clásica de Microsoft o desde el framework de .NET. Es posible entonces implementar funcionalidad conforme a Acme en forma directa en cualquier programa, o llegado el caso (mediante *wrapping*) exponer Acme como web service.

Interfaz gráfica – La versión actual de Acme soporta una variedad de *front-ends* de carácter gráfico, de los cuales he experimentado con tres. El ambiente primario, llamado AcmeStudio, es un entorno gráfico basado en Windows, susceptible de ser configurado para soportar visualizaciones específicas de estilos e invocación de herramientas auxiliares. Un segundo entorno, llamado Armani, utiliza Microsoft Visio como front-end gráfico y un back-end Java, que con alguna alquimia puede ser Microsoft Visual J++ o incluso Visual J# de .NET. Armani no es un entorno redundante sino, por detrás de la fachada de Visio, un lenguaje de restricción que extiende Acme basándose en lógica de predicados de primer orden, y que es por tanto útil para definir invariantes y heurísticas de estilos. Un tercer ambiente, más experimental, diseñado en ISI, utiliza sorprendentemente el editor de PowerPoint para manipulación gráfica acoplado con analizadores que reaccionan a cambios de una representación DCOM de los elementos arquitectónicos y de sus propiedades asociadas [GB99].

Generación de código – En los últimos años se ha estimado cada vez más deseable que un ADL pueda generar un sistema ejecutable, aunque más no sea de carácter prototípico. De tener que hacerlo manualmente, se podrían suscitar problemas de consistencia y trazabilidad entre una arquitectura y su implementación. Acme, al igual que Wright, se concibe como una notación de modelado y no proporciona soporte directo de generación de código. Por el contrario, diversos ADLs actuales pueden manejar Acme, incluidos Aesop, C2, SADL, UniCon y Wright.

Disponibilidad de plataforma – Ya me he referido a AcmeStudio (2.1), un front-end gráfico programado en Visual C++ y Java que corre en plataforma Windows y que proporciona un ambiente completo para diseñar modelos de arquitectura. La sección de Java requiere JRE, pero también se puede trabajar en términos de COM y Win32

ejecutando AcmeStudio.exe. Los otros dos ambientes gráficos (Armani y el entorno de ISI) son nativos de Windows e implementan intensivamente tecnología COM.

Paulatinamente, Armani se constituyó en un lenguaje de tipo ADL, especializado en la descripción de la estructura de un sistema y su evolución en el tiempo [Mon98]. Es un lenguaje puramente declarativo que describe la estructura del sistema y las restricciones a respetar, pero no hace referencia alguna a la generación del sistema o a la verificación de sus propiedades no funcionales o de consistencia. De alguna manera, la naturaleza de Armani captura también el *expertise* de los diseñadores, señalando su vinculación con la práctica de los patrones arquitectónicos, en este caso patrones de diseño. Armani se basa en siete entidades para describir las instancias del diseño: componentes, conectores, puertos, roles, sistemas, representaciones y propiedades. Para capturar las experiencias y las recetas o "reglas de pulgar", Armani implementa además otras seis entidades que son: tipos de elementos de diseño, tipos de propiedades, invariantes de diseño, heurísticas, análisis y estilos arquitectónicos. La sección denotacional de Armani tiene un aire de familia con la sintaxis de cláusulas de Horn en lenguaje Prolog.

El siguiente ejemplo ilustra la descripción de un modelo de tubería y filtros en Armani:

```
Style Pipe-and-Filter = {
	// Definir el vocabulario de diseño
	// Definir el tipo de flujo
	Property Type flowpathRecT = Record [ fromPt : string; toPt : string; ];
	// Definir tipos de puerto y rol
	Port Type inputT = { Property protocol : string = "char input"; };
	Port Type outputT = { Property protocol : string = "char output"; };
	Role Type sourceT = { Property protocol : string = "char source"; };
	Role Type sinkT = { Property protocol : string = "char sink"; };
	// Definir tipos de componentes
	Component Type Filter = {
		Port input : inputT;
		Port output : outputT;
		Property function : string;
		Property flowPaths : set{flowpathRecT}
			<< default : set{flowpathRecT} =
			[ fromPt : string = "input"; toPt : string = "output"]; >>;
			// restricción para limitar añadido de otras puertos
			Invariant forall p : port in self.Ports |
			satisfiesType(p, inputT) or satisfiesType(p,outputT);
	};
	// Definir tipos de componentes
	Connector Type Pipe = {
		Role source : sourceT;
		Role sink : sinkT;
		Property bufferSize : int;
		Property flowPaths : set{flowpathRecT} =
		[ from : string = "source"; to : string = "sink" ];
		// los invariantes requieren que un Pipe tenga
		// exactamente dos roles y un buffer con capacidad positiva.
		Invariant size(self.Roles) == 2;
		Invariant bufferSize >= 0;
```

```
};
// Definir diseño abstracto de análisis para todo el estilo
// y verificar ciclos en el grafo del sistema
Design Analysis hasCycle(sys :System) : boolean =
       forall c1 : Component in sys.Components | reachable(c1,c1);
       // definir análisis de diseño externo que computa la performance
       // de un componente
External Analysis throughputRate(comp :Component) : int =
armani.tools.rateAnalyses.throughputRate(comp);
// Especificar invariantes de diseño y heurística
// para sistemas construidos en este estilo.
// Vincular inputs a sinks y outputs a fuentes
Invariant forall comp : Component in self.Components |
Forall conn : Connector in self.Connectors |
Forall p : Port in comp.Ports |
Forall r : Role in conn.Roles |
       attached(p,r) ->
       ((satisfiesType(p,inputT) and satisfiesType(r,sinkT)) or
       (satisfiesType(p,outputT) and satisfiesType(r,sourceT)));
// no hay roles no asignados
Invariant forall conn : Connector in self.Connectors | forall r : Role in
conn.Roles |
exists comp : Component in self.Components | exists p : Port in
comp.Ports |
attached(p,r);
// flag de puertos no vinculados
Heuristic forall comp : Component in self.Components |
       forall p : Port in comp.Ports | exists conn : Connector in
       self.Connectors |
       exists r : Role in conn.Roles |
       attached(p,r);
// Restricción: En un estilo de tubería y filtros no puede haber ciclos
Invariant !hasCycle(self);
// Propiedad no funcional: Todos los componentes deben tener un
// throughput de 100 (unidades) como mínimo.
Heuristic forall comp : Component in self.Components |
       throughputRate(comp) >= 100;
}; // fin de la definición del estilo-familia.
```

Como puede verse, en tanto lenguaje Armani es transparentemente claro, aunque un tanto verboso. Una vez que se ha realizado la declaración del estilo, aún restaría agregar código para definir una instancia y especificar un conjunto de invariantes para limitar la evolución de la misma.

ADML

Como hubiera sido de esperarse ante la generalización del desarrollo en la era del Web, ADML (Architecture Description Markup Language) constituye un intento de estandarizar la descripción de arquitecturas en base a XML. Está siendo promovido desde el año 2000 por The Open Group y fue desarrollado originalmente en MCC. The Open Group ha sido también promotor de The Open Group Architectural Framework

(TOGAF). La página de cabecera de la iniciativa se encuentra en http://www.enterprise-architecture.info/Images/ADML/WEB%20ADML.htm.

Como quiera que sea, ADML agrega al mundo de los ADLs una forma de representación basada en estándares de la industria, de modo que ésta pueda ser leída por cualquier parser de XML. En ambientes Windows el *parser* primario y el serializador de XML se instala con Microsoft Internet Explorer de la versión 4 en adelante, y todas las aplicaciones de Office, así como SQL Server, poseen soporte nativo de XML y por lo tanto del lenguaje arquitectónico de *markup*. El Framework .NET de Microsoft incluye además clases (`xmlreader`, `xmlwriter`) que hacen que implementar tratamiento de documentos ADML, xADL, xArch y sus variantes resulte relativamente trivial. En consonancia con la expansión de Internet, ADML permite también definir vínculos con objetos externos a la arquitectura (fundamentación racional, diseños, componentes, etcétera), así como interactuar con diversos repositorios de industria, tales como las especificaciones de OASIS relativas a esquemas para SWIFT, IFX, OFX/OFE, BIPS, OTP, OMF, HL7, RosettaNet o similares.

ADML constituye además un tronco del que depende una cantidad de especificaciones más puntuales. Mientras ADML todavía reposaba en DTD (Document Type Definition), una sintaxis de metadata que ahora se estima obsoleta, las especificaciones más nuevas implementan esquemas extensibles de XML. La más relevante tal vez sea xADL (a pronunciar como "zaydal"), desarrollado por la Universidad de California en Irvine, que define XML Schemas para la descripción de familias arquitectónicas, o sea estilos. La especificación inicial de xADL 2.0 se encuentra en http://www.isr.uci.edu/projects/xarchuci/. Técnicamente xADL es lo que se llama una aplicación de una especificación más abstracta y genérica, xArch, que es un lenguaje basado en XML, elaborado en Irvine y Carnegie Mellon para la descripción de arquitecturas [DH01]. Cada tipo de conector, componente e interfaz de xADL incluye un *placeholder* de implementación que vendría a ser análogo a una clase virtual o abstracta de un lenguaje orientado a objetos. Éste es reemplazado por las variables correspondientes del modelo de programación y plataforma en cada implementación concreta. Esto permite vincular descripciones arquitectónicas y modelos directamente con cualquier binario, *scripting* o entidad en cualquier plataforma y en cualquier lenguaje.

Aesop

El nombre oficial es Aesop Software Architecture Design Environment Generator. Se ha desarrollado como parte del proyecto ABLE de la Universidad Carnegie Mellon, cuyo objetivo es la exploración de las bases formales de la arquitectura de software, el desarrollo del concepto de estilo arquitectónico y la producción de herramientas útiles a la arquitectura, de las cuales Aesop es precisamente la más relevante. La elaboración formal del proyecto ABLE, por otro lado, ha resultado en el lenguaje Wright, que en este estudio se trata separadamente. Uno de los mejores documentos sobre Aesop es el ensayo de David Garlan, Robert Allen y John Ockerbloom que explora el uso de estilos en el diseño arquitectónico [GAO94].

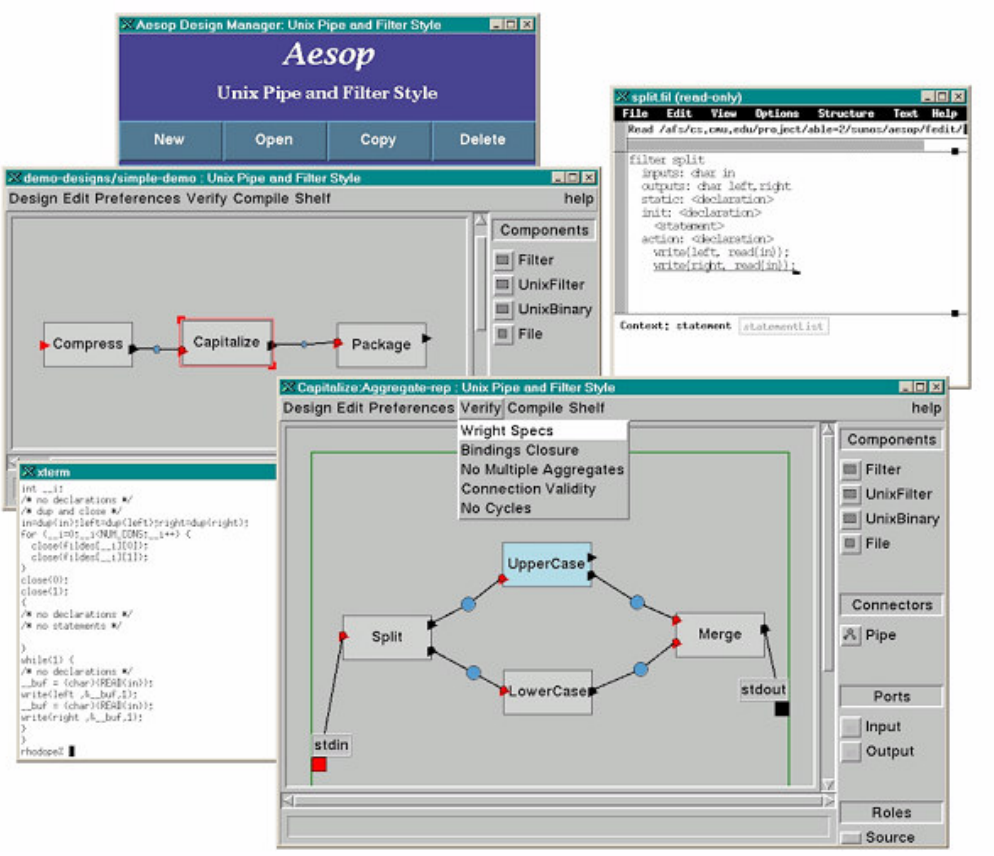


Fig. 2 - Ambiente gráfico de Aesop con diagrama de tubería y filtro

La definición también oficial de Aesop es “una herramienta para construir ambientes de diseño de software basada en principios de arquitectura”. El ambiente de desarrollo de Aesop System se basa en el estilo de tubería y filtros propio de UNIX. Un diseño en Aesop requiere manejar toda una jerarquía de lenguajes específicos, y en particular FAM Command Language (FCL, a pronunciar como “fickle”), que a su vez es una extensión de TCL orientada a soportar modelado arquitectónico. FCL es una combinación de TCL y C densamente orientada a objetos. En lo que respecta al manejo de métodos de análisis de tiempo real, Aesop implementa EDF (Earliest Deadline First).

Sitio de referencia – Razonablemente, se lo encuentra siguiendo el rastro de la Universidad Carnegie Mellon, la escuela de computación científica y el proyecto ABLE, en la página http://www-2.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html. Aunque hay bastante información en línea, el vínculo de distribución de Aesop estaba muerto o inaccesible en el momento de redacción de este documento (enero de 2004).

Estilos - En Aesop, conforme a su naturaleza orientada a objetos, el vocabulario relativo a estilos arquitectónicos se describe mediante la definición de sub-tipos de los tipos arquitectónicos básicos: Componente, Conector, Puerto, Rol, Configuración y Binding.

Interfaces - En Aesop (igual que en ACME y Wright) los puntos de interfaz se llaman puertos (*ports*).

Modelo semántico – Aesop presupone que la semántica de una arquitectura puede ser arbitrariamente distinta para cada estilo. Por lo tanto, no incluye ningún soporte nativo

para la descripción de la semántica de un estilo o configuración, sino que apenas presenta unos cuadros vacantes para colocar esa información como comentario.

Soporte de lenguajes - Aesop (igual que Darwin) sólo soporta nativamente desarrollos realizados en C++.

Generación de código – Aesop genera código C++. Aunque Aesop opera primariamente desde una interfaz visual, el código de Aesop es marcadamente más procedural que el de Acme, por ejemplo, el cual posee una estructura de orden más bien declarativo. El siguiente ejemplo muestra un estilo de tubería y filtros instanciado al tratamiento de un texto. Omito el código de encabezamiento básico del filtro:

```
// Primero corresponde definir el filtro
#include "filter_header.h"
void split(in,left,right)
{
char __buf;
int __i;
/* no hay declaraciones */
/* dup y cerrar */
in = dup(in);left = dup(left);
right = dup(right);
for (__i=0;__i<NUM_CONS;__i++) {
close(fildes[__i][0]);
close(fildes[__i][1]);
}
close(0);
close(1);
{
/*no hay declaraciones*/
/*no hacer nada*/
}
while(1)
{
/*no hay declaraciones*/
__buf = (char)((char) READ(in));
write(left,&__buf,1);
__buf = (char)((char) READ(in));
write(right,&__buf,1);
}
}
// Genera código para un sistema de estilo tubería y filtros
int main(int argc,char **argv) {
fable_init_event_system(&argc,argv,BUILD_PF); // inicializa eventos locales
fam_initialize(argc,argv); // inicializa la base de datos
arch_db = fam_arch_db::fetch(); // obtiene el puntero de la base de datos
t = arch_db.open(READ_TRANSACTION); // comienza transacción de lectura
fam_object o = get_object_parameter(argc,argv); // obtiene objeto raíz
if (!o.valid() || !o.type().is_type(pf_filter_type)) { // filtro no válido?
cerr << argv[0] << ": invalid parameter\n"; // si no es válido, cerrar
t.close();
fam_terminate();
```

```
exit(1);
}
pf_filter root = pf_filter::typed(o);
pf_aggregate ag = find_pf_aggregate(root); // encontrar agregado de la raiz
// (si no lo hay, imprimir diagnóstico; omito el código)
start_main(); // escribir el comienzo del main() generado
outer_io(root); // vincular puertos externas a stdin/out
if (ag.valid()) {
pipe_names(ag); // escribir código para conectar tuberías
bindings(root); // definir alias de bindings
spawn_filters(ag); // y hacer "fork off" de los filtros
}
finish_main(); // escribir terminación del main() generado
make_filter_header(num_pipes); // escribir header para los nombres de tuberías
t.close(); // cerrar transacción
fam_terminate(); // terminar fam
fable_main_event_loop(); // esperar el evento de terminación
fable_finish(); // y finalizar
return 0;
} // main
```

Disponibilidad de plataforma – Aesop no está disponible en plataforma Windows, aunque naturalmente puede utilizarse para modelar sistemas implementados en cualquier plataforma.

ArTek

ArTek fue desarrollado por Teknowledge. Se lo conoce también como ARDEC/Teknowledge Architecture Description Language. En opinión de Medvidovic no es un genuino ADL, por cuanto la configuración es modelada implícitamente mediante información de interconexión que se distribuye entre la definición de los componentes individuales y los conectores. En este sentido, aunque pueda no ser un ADL en sentido estricto, se le reconoce la capacidad de modelar ciertos aspectos de una arquitectura. De todas maneras, es reconocidamente un lenguaje específico de dominio y siempre fue presentado como un caso testigo de generación de un modelo a partir de una instancia particular de uso.

Disponibilidad de plataforma – Hoy en día ArTek no se encuentra disponible en ningún sitio y para ninguna plataforma.

C2 (C2 SADL, C2SADEL, xArch, xADL)

C2 o Chiron-2 no es estrictamente un ADL sino un estilo de arquitectura de software que se ha impuesto como estándar en el modelado de sistemas que requieren intensivamente pasaje de mensajes y que suelen poseer una interfaz gráfica dominante. C2 SADL (Simulation Architecture Description Language) es un ADL que permite describir arquitecturas en estilo C2. C2SADEL es otra variante; la herramienta de modelado canónica de este último es DRADEL (Development of Robust Architectures using a Description and Evolution Language). Llegado el momento del auge de XML, surge primero xArch y luego xADL, de los que ya se ha tratado en el apartado correspondiente a ADML y sus

derivaciones, pero sin hacer referencia a su conformidad con C2, que en los hechos ha sido enfatizado cada vez menos. Otra variante, SADL a secas, denota Structural Architecture Description Language; fue promovido alguna vez por SRI, pero no parece gozar hoy de buena salud.

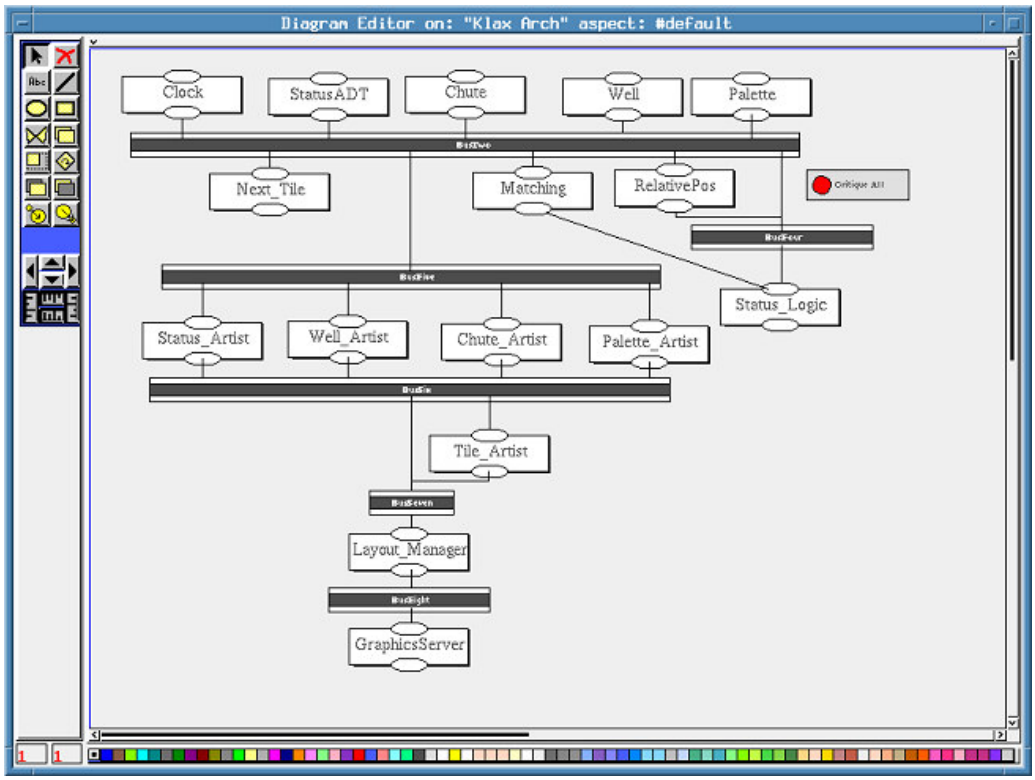


Fig. 3 - Ambiente de modelado C2 con diagrama de estilo

En una arquitectura de estilo C2, los conectores trasmiten mensajes entre componentes, los cuales mantienen el estado, ejecutan operaciones e intercambian mensajes con otros componentes a través de dos interfaces (llamadas *top* y *bottom*). Los componentes no intercambian mensajes directamente, sino a través de conectores. Cada interfaz de un componente puede vincularse con un solo conector, el cual a su vez se puede vincular a cualquier número de otros conectores o componentes. Los mensajes de requerimiento sólo se pueden enviar "hacia arriba" en la arquitectura, y los de notificación sólo "hacia abajo". La única forma de comunicación es a través de pasaje de mensajes, de modo que nunca se utiliza memoria compartida. Esta limitación se impone para permitir la independencia de sustratos, que en la jerga de C2 se refiere a la capacidad de reutilizar un componente en otro contexto. C2 no presupone nada respecto del lenguaje de programación en que se implementan los componentes o conectores (que pueden ser tranquilamente Visual Basic .NET o C#), ni sobre el manejo de *threads* de control de los componentes, el deployment de los mismos o el protocolo utilizado por los conectores. Por ello ha sido una de las bases de las modernas arquitecturas basadas en servicios.

Sitio de referencia – El sitio de ISR en la Universidad de California en Irvine está en http://www.isr.uci.edu/architecture/adl/SADL.html. xARch se encuentra en http://www.-isr.uci.edu/architecture/xarch/. XADL 2.0 se localiza en http://www.isr.uci.edu/projects-/xarchuci/.

Implementación de referencia – SADL se utilizó eficazmente para un sistema de control operacional de plantas de energía en Japón, implementado en Fortran 77. Se asegura que SADL permitió formalizar la arquitectura de referencia y asegurar su consistencia con la arquitectura de implementación.

Semántica – El modelo semántico de C2 es algo más primitivo que el de Rapide, por ejemplo. Los componentes semánticos se expresan en términos de relaciones causales entre mensajes de entrada y salida de una interfaz. Luego esta información se puede utilizar para rastrear linealmente una serie de eventos.

Soporte de lenguajes – C2 soporta desarrollos en C++, Ada y Java, pero en realidad no hay limitación en cuanto a los lenguajes propios de la implementación. Modelos de interoperabilidad de componentes como OLE y más recientemente COM+ no son ni perturbados ni reemplazados por C2, que los ha integrado con naturalidad [TNA+s/f].

En un ADL conforme a C2 de la generación SADL, el ejemplo de referencia de tubería y filtros se convertiría en un paso de mensajes entre componentes a través de una interfaz. Lo primero a definir sería un componente en función de una IDN (Interface Description Notation). La sintaxis sería como sigue:

```
component StackADT is
    interface
        top_domain
            in
                null;
            out
                null;
        bottom_domain
            in
                PushElement (value : stack_type);
                PopElement ();
                GetTopElement ();
            out
                ElementPushed (value : stack_type);
                ElementPopped (value : stack_type);
                TopStackElement (value : stack_type);
                StackEmpty ();
    parameters
            null;
    methods
            procedure Push (value : stack_type);
            function Pop () return stack_type;
            function Top () return stack_type;
            function IsEmpty () return boolean;
    behavior
        received_messages PushElement;
            invoke_methods Push;
            always_generate ElementPushed;
        received_messages PopElement;
            invoke_methods IsEmpty, Pop;
            always_generate StackEmpty xor ElementPopped;
        received_messages GetTopElement;
```

```
            invoke_methods IsEmpty, Top;
            always_generate StackEmpty xor TopStackElement;
    context
        top_most ADT
end StackADT;
```

Nótese que en este lenguaje, suficientemente legible como para que aquí prescindamos de comentarios, se trata un *stream* como un *stack*. Hasta aquí se han definido componentes, interfaces, métodos y conducta. En segundo término vendría la especificación declarativa de la arquitectura a través de ADN (Architecture Description Notation):

```
architecture StackVisualizationArchitecture is
    components
        top_most
            StackADT;
        internal
            StackVisualization1;
            StackVisualization2;
        bottom_most
            GraphicsServer;
    connectors
        connector TopConnector is
            message_filter no_filtering
        end TopConnector;
        connector BottomConnector is
            message_filter no_filtering
        end BottomConnector;
    architectural_topology
        connector TopConnector connections
            top_ports
                StackADT;
            bottom_ports
                StackVisualization1;
                StackVisualization2;
        connector BottomConnector connections
            top_ports
                StackVisualization1;
                StackVisualization2;
            bottom_ports
                GraphicsServer;
end StackVisualizationArchitecture;
system StackVisualizationSystem is
    architecture StackVisualizationArchitecture with
        StackADT is_bound_to IntegerStack;
        StackVisualization1 is_bound_to StackArtist1;
        StackVisualization2 is_bound_to StackArtist2;
        GraphicsServer is_bound_to C2GraphicsBinding;
end StackVisualizationSystem;
```

Por último, se podrían especificar imperativamente cambios en la arquitectura por medio de un tercer lenguaje, que sería un ACN (Architecture Construction Notation). Omitiremos esta ejemplificación. En algún momento C2 SADL (que nunca pasó de la

fase de prototipo alfa) fue reemplazado por C2SADEL, cuya sintaxis es diferente y ya no se articula en la misma clase de módulos declarativos e imperativos.

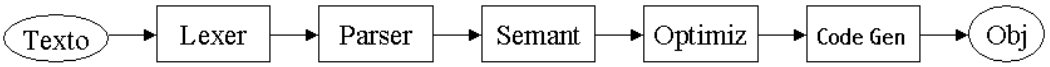
Disponibilidad de plataforma – Existen extensiones de Microsoft Visio para C2 y xADL. Una interfaz gráfica para C2SADEL disponible para Windows es DRADEL. También se puede utilizar una herramienta llamada SAAGE (disponible en el sitio de DASADA, http://www.rl.af.mil/tech/programs/dasada/tools/saage.html) que requiere explícitamente Visual J++, COM y ya sea Rational Rose o DRADEL como *front-ends*. Hay multitud de herramientas que generan lenguaje conforme al estándar C2, ya sea en estado crudo o, como es el caso en los proyectos más recientes, en formato xADL. Una de ellas, independientes de plataforma, es el gigantesco entorno gráfico ArchStudio, él mismo implementado en estilo C2. En una época ArchStudio incluía Argo, otro ambiente de diseño gráfico para C2 actualmente discontinuado.

CHAM

CHAM (Chemical Abstract Machine) no es estrictamente un ADL, aunque algunos autores, en particular Inverardi y Wolf [BB92] aplicaron CHAM para describir la arquitectura de un compilador. Se argumenta, en efecto, que CHAM proporciona una base útil para la descripción de una arquitectura debido a su capacidad de componer especificaciones para las partes y describir explícitamente las reglas de composición. Sin embargo, la formalización mediante CHAM es idiosincrática y (por así decirlo) hecha a mano, de modo que no hay criterios claros para analizar la consistencia y la completitud de las descripciones de configuración. Convendrá contar entonces con alguna herramienta de verificación.

CHAM es una técnica de especificación basada en álgebra de procesos que utiliza como fundamento teórico los sistemas de rescritura de términos para capturar la conducta comunicativa de los componentes arquitectónicos. El modelo de CHAM reposa en una metáfora química en la cual la conducta de una arquitectura se especifica definiendo moléculas y soluciones de moléculas. Las moléculas constituyen los componentes básicos, mientras que las soluciones son multiconjuntos de moléculas que definen los estados de una CHAM. Una especificación CHAM también contiene reglas de transformación que dictan las formas en que pueden evolucionar las soluciones (o sea, en que pueden cambiar los estados). En un momento dado, se puede aplicar un número arbitrario de reglas a una solución, siempre que no entren en conflicto. De esta forma es posible modelar conductas paralelas, ejecutando transformaciones en paralelo. Cuando se aplica más de una regla a una molécula o conjunto, CHAM efectúa una decisión no determinista, escogiendo alguna de las reglas definidas.

A fin de ilustrar un modelo en CHAM relacionado con nuestro ejemplo canónico de tubería y filtros, supongamos que se desea compilar una pieza de código en Lisp semejante a la que se encuentra en el Software Development Kit que acompaña a Visual Studio.NET, el cual incluye un programa en C# para generar un compilador Lisp y sus respectivas piezas. Estas piezas están articuladas igual que en el ejemplo, incluyendo un Lexer, un Parser, un Semantizador y un Generador de código. A través de las sucesivas transformaciones operadas en los filtros, el programa fuente se convierte en código objeto en formato MSIL.



La especificación del modelo CHAM correspondiente es altamente intuitiva. Se comienza con un conjunto de constantes *P* que representan los elementos de procesamiento, un conjunto de constantes *D* que denotan los elementos de dato y un operador infijo *P* que expresa el estado de un elemento. Los elementos de conexión están dados por un tercer conjunto *C* consistente en dos operaciones, *i* y *o*, que actúan sobre los elementos de dato y que obviamente denotan entrada y salida. Lo que se llama sintaxis *sigma* de moléculas en esta arquitectura secuencial es entonces:

- `M ::= P | C | M <> M`
- `P ::= text | lexer | parser | semantor | optimizer | generator`
- `D ::= char | tok | phr | cophr | obj`
- `C ::= i(D) | o(D)`

El siguiente paso en CHAM es siempre definir una solución inicial S_1. Esta solución es un subconjunto de todas las moléculas posibles que se pueden construir bajo *sigma* y corresponde a la configuración inicial conforme a la arquitectura.

```
S1 = text <> o(char), i(char) <> o(tok) <> lexer
       i(tok) <> o(phr) <> parser, i(phr) <> o(cophr) <> semantor,
       i(cophr) <> o(cophr) <> optimizer, i(cophr) <> o(obj) <> generator
```

Esto quiere decir que se ingresa inicialmente texto en forma de caracteres, el Lexer lo transforma en *tokens*, el Parser en frases, el Semantor en co-frases que luego se optimizan, y el Generador de código lo convierte en código objeto. El paso final es especificar tres reglas de transformación:

```
T1 = text <> o(char) --> o(char) <> text
T2 = i(d) <> m1, o(d) <> m2 --> m1 <> i(d), m2 <> o(d)
T3 = o(obj) <> generator <> i(cophr) --> i(char) <> o(tok) <> lexer,
       i(tok) <> o(phr) <> parser, i(phr) <> o(cophr) <> semantor,
       i(cophr) <> o(cophr) <> optimizer, i(cophr) <> o(obj) <> generator
```

La primera regla hace que el código fuente quede disponible para compilar. El resto puede interpretarse fácilmente a través de las correspondencias entre las tres series de notaciones.

Disponibilidad de plataforma – CHAM es un modelo de máquina abstracta independiente de plataforma y del lenguaje o paradigma de programación que se vaya a utilizar en el sistema que se modela. En una reciente tesis de licenciatura de la Facultad de Ciencias Exactas de la Universidad de Buenos Aires, de hecho, se utilizó CHAM (y no cálculo JI) para modelar las prestaciones de XLANG del middleware de integración BizTalk de Microsoft. Aunque CHAM no es decididamente un ADL, nos ha parecido que su notación y su fundamentación subyacente son una contribución interesante para una mejor comprensión de las herramientas descriptivas de una arquitectura. En el ejemplo propuesto se discierne no sólo un estilo en toda su pureza, sino que se aprecia además la

potencia expresiva de un lenguaje de descripción arquitectónica y un patrón de diseño que se impone a la hora de definir el modelo de cualquier compilador secuencial.

Darwin

Darwin es un lenguaje de descripción arquitectónica desarrollado por Jeff Magee y Jeff Kramer [MEDK95, MK96]. Darwin describe un tipo de componente mediante una interfaz consistente en una colección de servicios que son ya sea provistos (declarados por ese componente) o requeridos (o sea, que se espera ocurran en el entorno). Las configuraciones se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios.

Darwin soporta la descripción de arquitecturas que se reconfiguran dinámicamente a través de dos construcciones: instanciación tardía [*lazy*] y construcciones dinámicas explícitas. Utilizando instanciación laxa, se describe una configuración y se instancian componentes sólo en la medida en que los servicios que ellos provean sean utilizados por otros componentes. La estructura dinámica explícita, en cambio, se realiza mediante constructos de configuración imperativos. De este modo, la declaración de configuración deviene un programa que se ejecuta en tiempo de ejecución, antes que una declaración estática de la estructura.

Cada servicio de Darwin se modeliza como un nombre de canal, y cada declaración de *binding* es un proceso que trasmite el nombre del canal al componente que requiere el servicio. En una implementación generada en Darwin, se presupone que cada componente primitivo está implementado en algún lenguaje de programación, y que para cada tipo de servicio se necesita un ligamento (*glue*) que depende de cada plataforma. El algoritmo de elaboración actúa, esencialmente, como un servidor de nombre que proporciona la ubicación de los servicios provistos a cualquier componentes que se ejecute. El ejemplo muestra la definición de una tubería en Darwin [MK96: 5].

```
component pipeline (int n) {
       provide input;
       require output;
       array F[n]:filter;
       forall k:0..n-1 {
              inst F[k];
              bind F[k].output -- output;
              when k<n-1 bind
                     F[k].next -- F[k+1].prev;
              }
              bind
                     input -- F[0].prev;
                     F[n-1].next -- output;
              }
```

Darwin no proporciona una base adecuada para el análisis de la conducta de una arquitectura, debido a que el modelo no dispone de ningún medio para describir las propiedades de un componente o de sus servicios más que como comentario. Los componentes de implementación se interpretan como cajas negras, mientras que la

colección de tipos de servicio es una colección dependiente de plataforma cuya semántica tampoco se encuentra interpretada en el *framework* de Darwin [All97].

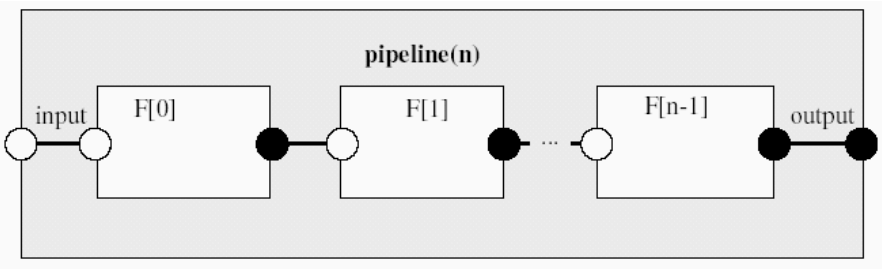


Fig. 4 - Diagrama de tubería en Darwin

Objetivo principal – Como su nombre lo indica, Darwin está orientado más que nada al diseño de arquitecturas dinámicas y cambiantes.

Estilos - El soporte de Darwin para estilos arquitectónicos se limita a la descripción de configuraciones parametrizadas, como la del ejemplo de la tubería que figura más arriba. Esta descripción, en particular, indica que una tubería es una secuencia lineal de filtros, en la que la salida de cada filtro se vincula a la entrada del filtro siguiente en la línea. Un estilo será entonces expresable en Darwin en la medida en que pueda ser constructivamente caracterizado; en otras palabras, para delinear un estilo hay que construir un algoritmo capaz de representar a los miembros de un estilo. Dada su especial naturaleza, es razonable suponer que Darwin se presta mejor a la descripción de sistemas que poseen características dinámicas.

Interfaces – En Darwin las interfaces de los componentes consisten en una colección de servicios que pueden ser provistos o requeridos.

Conectores – Al pertenecer a la clase en la que Medvidovic [Med96] agrupa a los lenguajes de configuración *in-line*, en Darwin (al igual que en Rapide) no es posible ponerle nombre, sub-tipear o reutilizar un conector. Tampoco se pueden describir patrones de interacción independientemente de los componentes que interactúan.

Semántica - Darwin proporciona una semántica para sus procesos estructurales mediante el cálculo Л. Cada servicio se modeliza como un nombre de canal, y cada declaración de enlace (*binding*) se entiende como un proceso que trasmite el nombre de ese canal a un componente que requiere el servicio. Este modelo se ha utilizado para demostrar la corrección lógica de las configuraciones de Darwin. Dado que el cálculo Л ha sido designado específicamente para procesos móviles, su uso como modelo semántico confiere a las configuraciones de Darwin un carácter potencialmente dinámico. En un escenario como el Web, en el que las entidades que interactúan no están ligadas por conexiones fijas ni caracterizadas por propiedades definidas de localización, esta clase de cálculo se presenta como un formalismo extremadamente útil. Ha sido, de hecho, una de las herramientas formales que estuvo en la base de los primeros modelos del XLANG de Microsoft, que ahora se encuentra derivando hacia BPEL4WS (Business Process Execution Language for Web Services).

Análisis y verificación – A pesar del uso de un modelo de cálculo Л para las descripciones estructurales, Darwin no proporciona una base adecuada para el análisis del comportamiento de una arquitectura. Esto es debido a que el modelo no posee

herramientas para describir las propiedades de un componente o de los servicios que presta. Las implementaciones de un componente vendrían a ser de este modo cajas negras no interpretadas, mientras que los tipos de servicio son una colección dependiente de la plataforma cuya semántica también se encuentra sin interpretar en el framework de Darwin.

Interfaz gráfica – Darwin proporciona notación gráfica. Existe también una herramienta gráfica (Software Architect's Assistant) que permite trabajar visualmente con lenguaje Darwin. El desarrollo de SAA parecería estar discontinuado y ser fruto de una iniciativa poco formal, lo que sucede con alguna frecuencia en el terreno de los ADLs.

Soporte de lenguajes – Darwin (igual que Aesop) soporta desarrollos escritos en C++, aunque no presupone que los componentes de un sistema real estén programados en algún lenguaje en particular.

Observaciones – Darwin (lo mismo que UniCon) carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos de servicio predefinidos. Darwin presupone que la colección de tipos de servicio es suministrada por la plataforma para la cual se desarrolla una implementación, y confía en la existencia de nombres de tipos de servicio que se utilizan sin interpretación, sólo verificando su compatibilidad.

Disponibilidad de plataforma – Aunque el ADL fue originalmente planeado para ambientes tan poco vinculados al modelado corporativo como hoy en día lo es Macintosh, en Windows se puede modelar en lenguaje Darwin utilizando Software Architect's Assistant. Esta aplicación requiere JRE. Se la puede descargar en forma gratuita desde http://www.doc.ic.ac.uk/~kn/java/saaj.html

Jacal

Es un lenguaje de descripción de arquitecturas de software de propósito general creado en la Universidad de Buenos Aires, por un grupo de investigación del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales.

Objetivo principal – El objetivo principal de Jacal es lo que actualmente se denomina "animación" de arquitecturas. Esto es, poder visualizar una simulación de cómo se comportaría en la práctica un sistema basado en la arquitectura que se ha representado.

Más allá de este objetivo principal, el diseño de Jacal contempla otras características deseables en un ADL, como por ejemplo contar con una representación gráfica que permita a simple vista transmitir la arquitectura del sistema, sin necesidad de recurrir a información adicional. Para este fin, se cuenta con un conjunto predefinido (extensible) de conectores, cada uno con una representación distinta.

Sitio de referencia – http://www.dc.uba.ar/people/profesores/nicok/jacal.htm..

Estilos – Jacal no cuenta con una notación particular para expresar estilos, aunque por tratarse de un lenguaje de propósito general, puede ser utilizado para expresar arquitecturas de distintos estilos. No ofrece una forma de restringir una configuración a un estilo específico, ni de validar la conformidad.

Interfaces – Cada componente cuenta con puertos (*ports*) que constituyen su interfaz y a los que pueden adosarse conectores.

Semántica – Jacal tiene una semántica formal que está dada en función de redes de Petri. Se trata de una semántica denotacional que asocia a cada arquitectura una red correspondiente. La semántica operacional estándar de las redes de Petri es la que justifica la animación de las arquitecturas.

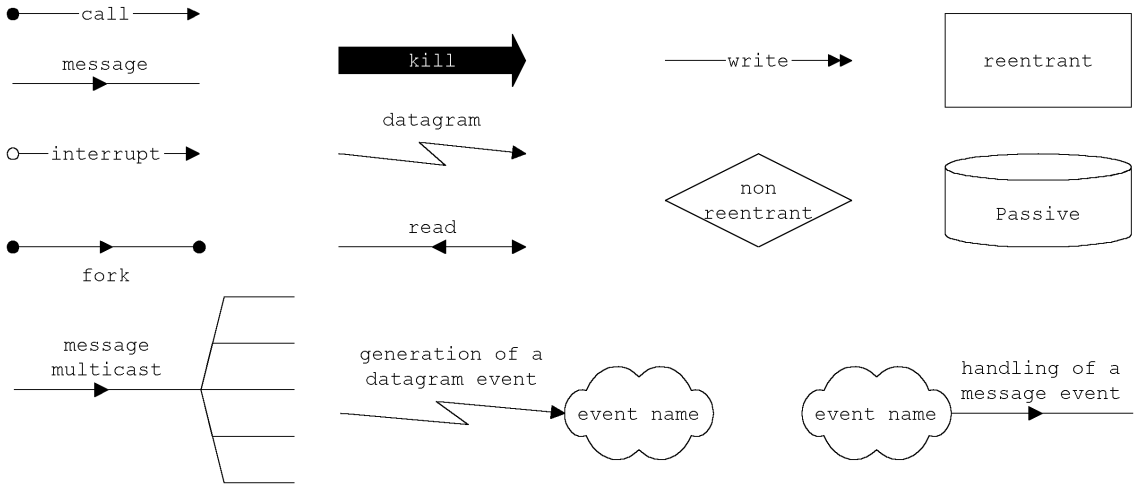
Además del nivel de interfaz, que corresponde a la configuración de una arquitectura ya que allí se determina la conectividad entre los distintos componentes, Jacal ofrece un nivel de descripción adicional, llamado nivel de comportamiento. En este nivel, se describe la relación entre las comunicaciones recibidas y enviadas por un componente, usando diagramas de transición de estados con etiquetas en los ejes que corresponden a nombres de puertos por los que se espera o se envía un mensaje.

Análisis y verificación– Las animaciones de arquitecturas funcionan como casos de prueba. La herramienta de edición y animación disponible en el sitio del proyecto permite dibujar arquitecturas mediante un editor orientado a la sintaxis, para luego animarlas y almacenar el resultado de las ejecuciones en archivos de texto. Esta actividad se trata exclusivamente de una tarea de testing, debiendo probarse cada uno de los casos que se consideren críticos, para luego extraer conclusiones del comportamiento observado o de las trazas generadas. Si bien no se ofrecen actualmente herramientas para realizar procesos de verificación automática como modelchecking, la traducción a redes de Petri ofrece la posibilidad de aplicar al resultado otras herramientas disponibles en el mercado.

Interface gráfica – Como ya se ha dicho, la notación principal de Jacal es gráfica y hay una herramienta disponible en línea para editar y animar visualmente las arquitecturas.

En el nivel de interfaz, existen símbolos predeterminados para representar cada tipo de componente y cada tipo de conector, como se muestra en la siguiente figura.

A continuación, se da una explicación informal de la semántica de cada uno de los tipos de conectores mostrados en la figura.



`call`: transfiere el control y espera una respuesta.

`message`: coloca un mensaje en una cola y sigue ejecutando.

`interrupt`: interrumpe cualquier flujo de control activo en el receptor y espera una respuesta.

`fork`: agrega un flujo de control al receptor, el emisor continúa ejecutando.

`kill`: detiene todos los flujos de control en el receptor, el emisor continúa ejecutando.

`datagram`: si el receptor estaba esperando una comunicación por este puerto, el mensaje es recibido; de lo contrario, el mensaje se pierde; en cualquier caso el emisor sigue ejecutando.

`read`: la ejecución en el emisor continúa, de acuerdo con el estado del receptor.

`write`: cambia el estado del receptor, el emisor continúa su ejecución.

Generación de código – En su versión actual, Jacal no genera código de ningún lenguaje de programación, ya que no fuerza ninguna implementación única para los conectores. Por ejemplo, un conector de tipo `message` podría implementarse mediante una cola de alguna plataforma de middleware (como MSMQ o MQSeries) o directamente como código en algún lenguaje. No obstante, la herramienta de edición de Jacal permite exportar a un archivo de texto la estructura estática de una arquitectura, que luego puede ser convertida a código fuente para usar como base para la programación.

Disponibilidad de plataforma – La herramienta que actualmente está disponible para editar y animar arquitecturas en Jacal es una aplicación Win32, que no requiere instalación, basta con copiar el archivo ejecutable para comenzar a usarla.

El ambiente consiste en una interfaz gráfica de usuario, donde pueden dibujarse representaciones Jacal de sistemas, incluyendo tanto el nivel de interfaz como el de comportamiento. Se pueden editar múltiples sistemas simultáneamente y, abriendo distintas vistas, visualizar simultáneamente los dos niveles de un mismo sistema, para uno o más componentes.

El editor es orientado a la sintaxis, en el sentido de que no permite dibujar configuraciones inválidas. Por ejemplo, valida la compatibilidad entre el tipo de un componente y los conectores asociados. Para aumentar la flexibilidad (especialmente en el orden en que se dibuja una arquitectura), se dejan otras validaciones para el momento de la animación.

Cuando se anima una arquitectura, los flujos de control se generan en comunicaciones iniciadas por el usuario (representa una interacción con el mundo exterior) haciendo clic en el extremo de origen de un conector que no esté ligado a ningún puerto. Los flujos de control se muestran como círculos de colores aleatorios que se mueven a lo largo de los conectores y las transiciones. Durante la animación, el usuario puede abrir vistas adicionales para observar el comportamiento interno de uno o más componentes, sin dejar de tener en pantalla la vista global del nivel de interfaz.

La siguiente figura muestra la interfaz de la aplicación con un caso de estudio.

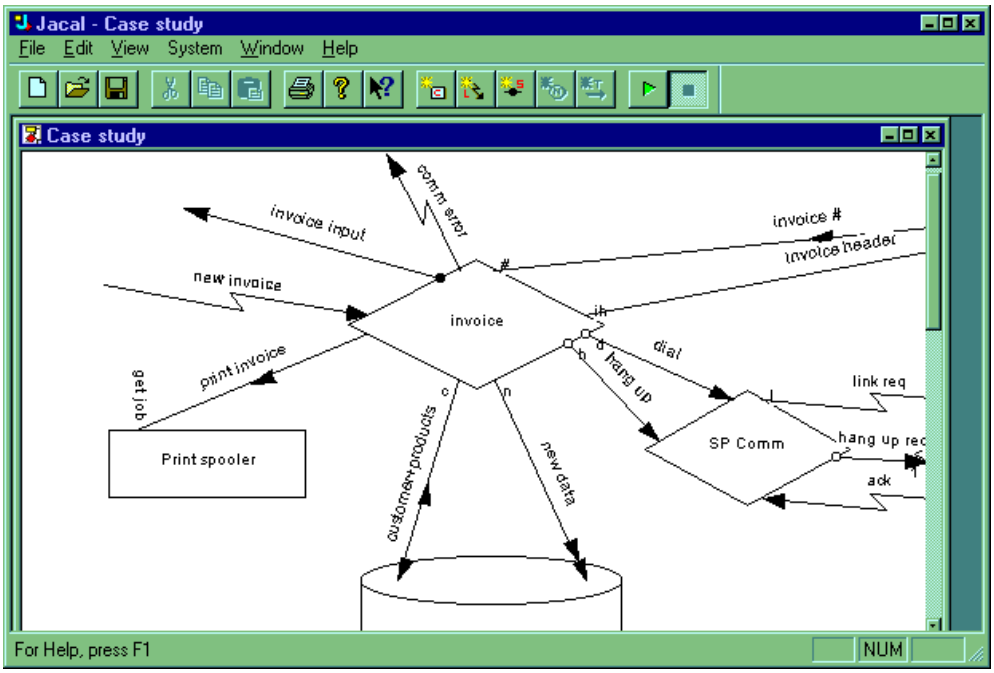


Fig. 5 – Estudio de caso en Jacal

Actualmente se encuentra en desarrollo una nueva versión del lenguaje y de la aplicación (Jacal 2). En esta nueva versión, se utilizará Microsoft Visio como interfaz del usuario, tanto para dibujar como para animar arquitecturas. La fecha estimada para su publicación es mediados de 2004.

LILEANNA

Al igual que en el caso de ArTek, en opinión de Medvidovic LILEANNA no es un genuino ADL, por cuanto la configuración es modelizada implícitamente mediante información de interconexión que se distribuye entre la definición de los componentes individuales y los conectores. En este sentido, aunque pueda no ser un ADL en sentido estricto, se le reconoce la capacidad de modelizar ciertos aspectos de una arquitectura.

LILEANNA es, visto como ADL, estructural y sintácticamente distinto a todos los demás. De hecho, es oficialmente un lenguaje de interconexión de módulos (MIL), basado en expresiones de módulo propias de la programación parametrizada. Un MIL se puede utilizar descriptivamente, para especificar y analizar un diseño determinado, o constructivamente, para generar un nuevo sistema en base a módulos preexistentes, ejecutando el diseño. Típicamente, la programación parametrizada presupone la disponibilidad de dos clases de bibliotecas, que contienen respectivamente expresiones de módulo (que describen sistemas en términos de interconexiones de módulos) y grafos de módulo (que describen módulos y relaciones entre ellos, y que pueden incluir código u otros objetos de software).

LILEANNA es un ADL (o más estrictamente un MIL) que utiliza el lenguaje Ada para la implementación y Anna para la especificación. Fue desarrollado como parte del proyecto DSSA ADAGE, patrocinado por ARPA. La implementación fue llevada a cabo por Will

Tracz de Loral Federal Systems y se utilizó para producir software de navegación de helicópteros.

La semántica formal de LILEANNA se basa en la teoría de categorías, siguiendo ideas desarrolladas para el lenguaje de especificación Clear; posteriormente se le agregó una semántica basada en teoría de conjuntos. Las propiedades de interconexión de módulos se relacionan bastante directamente con las de los componentes efectivos a través de la semántica de las expresiones de módulo. Aunque una especificación en LILEANNA es varios órdenes de magnitud más verbosa de lo que ahora se estima deseable para visualizar una descripción, incluye un "editor de *layout*" gráfico basado en "una notación como la que usan típicamente los ingenieros, es decir cajas y flechas". Es significativo que el documento de referencia más importante sobre el proyecto [Gog96], tan puntilloso respecto de las formas sintácticas y los métodos formales concomitantes a la programación parametrizada, se refiera a su representación visual en estos términos.

Aunque parecería corresponderse con un paradigma peculiar de programación, el modelo parametrizado de LILEANNA soporta diferentes estilos de comunicación, tales como variables compartidas, tuberías, paso de mensajes y *blackboarding*. Mediante un sistema auxiliar llamado TOOR, se pudo implementar además el rastreo (*tracing*) de dependencias entre objetos potencialmente evolutivos y relaciones entre objetos en función del tiempo. Esta estrategia, pomposamente llamada hiper-requerimiento, se funda en métodos bien conocidos en programación parametrizada e hiperprogramación. En este contexto, "hiper" connota una semántica de vinculación con el mundo exterior análoga a la de la idea de hipertexto. La hiperprogramación y los hiper-requerimientos soportan reutilización. TOOR, la hiper-herramienta agregada a LILEANNA, está construida a imagen y semejanza de FOOPS, un lenguaje orientado a objetos de propósito general. TOOR utiliza módulos semejantes a los de FOOPS para declarar objetos de software y relaciones, y luego generar vínculos a medida que los objetos se interconectan y evolucionan. TOOR proporciona facilidades de hipermedia basados en HTML, de modo que se pueden agregar gráficos, grafos y videos, así como punteros a documentos tradicionales.

Dado que el modelo de programación parametrizada subyacente a LILEANNA (que habla de teorías, axiomas, grafos, vistas, *stacks*, aserciones, estructuras verticales y horizontales, *packages* y máquinas virtuales en un sentido idiosincrático a ese paradigma) no detallaré aquí las peculiaridades del lenguaje. Los fundamentos formales de LILEANNA son poderosos, pero no están en línea con el estilo de arquitectura orientada a servicios o con modelos igualmente robustos, como los de C2 y Wright. Aunque no pueda esperarse que LILEANNA tenga en el futuro próximo participación directa y protagónica en la corriente principal de los ADLs, encarna algunas ideas sobre lo que se puede hacer con un lenguaje descriptivo que pueden resultar más que informativas para los arquitectos de software.

MetaH/AADL

Así como LILEANNA es un ADL ligado a desarrollos que guardan relación específica con helicópteros, MetaH modela arquitecturas en los dominios de guía, navegación y control (GN&C) y en el diseño aeronáutico. Aunque en su origen estuvo ligado

estrechamente a un dominio, los requerimientos imperantes obligaron a implementar recursos susceptibles de extrapolarse productivamente a la tecnología de ADLs de propósito general. AADL (Avionics Architecture Description Language) está basado en la estructura textual de MetaH. Aunque comparte las mismas siglas, no debe confudirse este AADL con Axiomatic Architecture Description Language, una iniciativa algo más antigua que se refiere al diseño arquitectónico físico de computadoras paralelas.

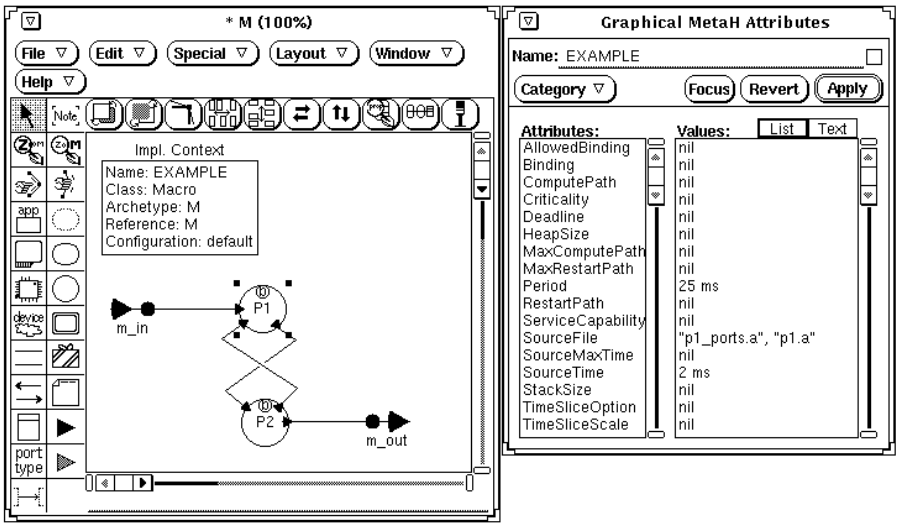


Fig. 6 - Ambiente gráfico MetaH con diagrama de macro

Sitio de referencia - http://www.htc.honeywell.com/metah/

Objetivo principal – MetaH ha sido diseñado para garantizar la puesta en marcha, la confiabilidad y la seguridad de los sistemas modelados, y también considera la disponibilidad y las propiedades de los recursos de hardware.

Soporte de lenguajes – MetaH está exclusivamente ligado a desarrollos hechos en Ada en el dominio de referencia.

Disponibilidad de plataforma – Para trabajar con MetaH en ambientes Windows, Honeywell proporciona un MetaH Graphical Editor implementado en DoME, que provee un conjunto extenso de herramientas visuales y de edición de texto.

Rapide

Se puede caracterizar como un lenguaje de descripción de sistemas de propósito general que permite modelar interfaces de componentes y su conducta observable. Sería tanto un ADL como un lenguaje de simulación. La estructura de Rapide es sumamente compleja, y en realidad articula cinco lenguajes: el lenguaje de tipos describe las interfaces de los componentes; el lenguaje de arquitectura describe el flujo de eventos entre componentes; el lenguaje de especificación describe restricciones abstractas para la conducta de los componentes; el lenguaje ejecutable describe módulos ejecutables; y el lenguaje de patrones describe patrones de los eventos. Los diversos sub-lenguajes comparten la misma visibilidad, *scoping* y reglas de denominación, así como un único modelo de ejecución.

Sitio de referencia – Universidad de Stanford - http://pavg.stanford.edu/rapide/

Objetivo principal – Simulación y determinación de la conformidad de una arquitectura.

Interfaces – En Rapide los puntos de interfaz de los componentes se llaman constituyentes.

Conectores – Siendo lo que Medvidovic (1996) llama un lenguaje de configuración *in-line*, en Rapide (al igual que en Darwin) no es posible poner nombre, sub-tipear o reutilizar un conector.

Semántica – Mientras muchos lenguajes de tipo ADL no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, Wright y Rapide permiten modelar la conducta de sus componentes. Rapide define tipos de componentes (llamados interfaces) en términos de una colección de eventos de comunicación que pueden ser *observados* (acciones externas) o *iniciados* (acciones públicas). Las interfaces de Rapide definen el comportamiento computacional de un componente vinculando la observación de acciones externas con la iniciación de acciones públicas. Cada especificación posee una conducta asociada que se define a través de conjuntos de eventos parcialmente ordenados (*posets*); Rapide utiliza patrones de eventos para identificar *posets*, de manera análoga a la del método `match` de las expresiones regulares de .NET Framework. Para describir comportamientos Rapide también implementa un lenguaje cuyo modelo de interfaz se basa en Standard ML, extendido con eventos y patrones de eventos.

Análisis y verificación automática – En Rapide, el monitoreo de eventos y las herramientas nativas de filtrado facilitan el análisis de arquitectura. También es posible implementar verificación de consistencia y análisis mediante simulación. En esencia, en Rapide toda la arquitectura es simulada, generando un conjunto de eventos que se supone es compatible con las especificaciones de interfaz, conducta y restricciones. La simulación es entonces útil para detectar alternativas de ejecución. Rapide también proporciona una caja de herramientas específica para simular la arquitectura junto con la ejecución de la implementación. Sin embargo (como ha señalado Medvidovic en su *survey*) un proceso de ejecución solamente provee una idea del comportamiento con un juego particular de variables (un *poset* en particular), antes que una confirmación de la conducta frente a todos los valores y escenarios posibles. Esto implica que una corrida del proceso de simulación simplemente testea la arquitectura, y no proporciona un análisis exhaustivo del escenario. Nada garantiza que no pueda surgir una inconsistencia en una ejecución diferente. En general, la arquitectura de software mantiene una actitud de reserva crítica frente a la simulación. Escribe Paul Clements: “La simulación es inherentemente una herramienta de validación débil en la medida que sólo presenta una sola ejecución del sistema cada vez; igual que el *testing*, sólo puede mostrar la presencia antes que la ausencia de fallas. Más poderosos son los verificadores o probadores de teoremas que son capaces de comparar una aserción de seguridad contra todas las posibbles ejecuciones de un programa simultáneamente” [Cle96].

Interfaz gráfica – Rapide soporta notación gráfica.

Soporte de lenguajes – Rapide soporta construcción de sistemas ejecutables especificados en VHDL, C. C++, Ada y Rapide mismo. El siguiente es un ejemplo próximo a nuestro

caso de referencia de tubería y filtros, sólo que en este caso es bidireccional, ya que se ha definido una respuesta de notificación. Rapide no contempla estilos de la misma manera que la mayor parte de los otros ADLs. Nótese que Rapide establece tres clases de conexiones; el conector correspondiente a una conexión de tipo *pipe* es =>

```
type Producer (Max : Positive) is interface
     action out Send (N: Integer);
     action in Reply(N : Integer);
behavior
     Start => send(0);
     (?X in Integer) Reply(?X) where ?X<Max => Send(?X+1);
end Producer;
type Consumer is interface
     action in Receive(N: Integer);
     action out Ack(N : Integer);
behavior
    (?X in Integer) Receive(?X) => Ack(?X);
end Consumer
architecture ProdCon() return SomeType is
     Prod : Producer(100); Cons : Consumer;
connect
     (?n in Integer) Prod.Send(?n) => Cons.Receive(?n);
     Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCon;
```

Generación de código – Rapide puede generar código C, C++ y Ada.

Observaciones – En materia de evolución y soporte de sub-tipos, Rapide soporta herencia, análoga a la de los lenguajes OOP.

Implementación de referencia – Aunque se ha señalado su falta de características de escalabilidad, Rapide se ha utilizado en diversos proyectos de gran escala. Un ejemplo representativo es el estándar de industria X/Open Distributed Transaction Processing.

Disponibilidad de plataforma – Rapide ha desarrollado un conjunto de herramientas que sólo se encontraba disponible para Solaris 2.5, SunOS 4.1.3. y Linux. Este *toolset* no ha evolucionado desde 1997, y tampoco ha avanzado más allá de la fase de prototipo.

UML - De OMT al Modelado OO

UML forma parte del repertorio conocido como lenguajes semi-formales de modelado. Esta variedad de herramientas se remonta a una larga tradición que arrancó a mediados de la década de 1970 con PSL/PSA, SADT y el análisis estructurado. Alrededor de 1990 aparecieron los primeros lenguajes de especificación orientados a objeto propuestos por Grady Booch, Peter Coad, Edward Yourdon y James Rumbaugh. A instancias de Rumbaugh, Booch e Ivar Jacobson, finalmente, estos lenguajes se orientaron hacia lo que es hoy UML (Unified Modeling Language), que superaba la incapacidad de los primeros lenguajes de especificación OO para modelar aspectos dinámicos y de comportamiento de un sistema introduciendo la noción de casos de uso. De hecho, UML surgió de la convergencia de los métodos de Booch orientados a comportamiento, la Object Modeling Technique (OMT) de Rumbaugh y el OOSE/Objectory de Jacobson, así como de otras

tecnologías tales como los gráficos de estado de Jarel, los patrones de documentación de Gamma y otros formalismos.

En términos de número, la masa crítica de los conocedores de UML no admite comparación con la todavía modesta población de especialistas en ADLs. En la comunidad de arquitectos existen dos facciones claramente definidas; la primera, vinculada con el mundo de Rational y UML, impulsa el uso casi irrestricto de UML como si fuera un ADL normal; la segunda ha señalado reiteradas veces las limitaciones de UML no sólo como ADL sino como lenguaje universal de modelado. La literatura crítica de UML es ya un tópico clásico de la reciente arquitectura de software. Entre las deficiencias que se han señalado de UML como lenguaje de especificación están las siguientes:

(1) Un caso de uso de UML no puede especificar los requerimientos de interacción en situaciones en las que un sistema deba iniciar una interacción entre él mismo y un actor externo, puesto que proscribe asociaciones entre actores [Gli00].

(2) Al menos en algunas especificaciones, como UML 1.3, es imposible expresar relaciones secuenciales, paralelas o iterativas entre casos de uso, salvo mediante extensiones o estereotipos que introducen oscuridad en el diseño del sistema; UML, efectivamente, prohíbe la descomposición de casos de uso y la comunicación entre ellos.

(3) UML tampoco puede expresar una estructura entre casos de uso ni una jerarquía de casos de una forma fácil y directa; una precondición tal como "el caso de uso A requiere que el caso de uso X haya sido ejecutado previamente" no se puede expresar formalmente en ese lenguaje.

(4) Un modelo de casos de uso tampoco puede expresar adecuadamente la conducta de un sistema dependiente de estados, o la descomposición de un sub-sistema distribuido.

(5) El modelado del flujo de información en un sistema consistente en subsistemas es al menos confuso en UML, y no se puede expresar en una sola vista [Gli00].

(6) Otros autores, como Theodor Tempelmeier [Tem99, Tem01] han opuesto objeciones a UML como herramienta de diseño en el caso de sistemas embebidos, de alta concurrencia o de misión crítica y/o tiempo real, así como han señalado su inutilidad casi total para modelar sistemas con cualidades emergentes o sistemas que involucren representación del conocimiento.

(7) Otros autores dedicaron tesis enteras a las dificultades de UML referidas a componentes reflexivos y aspectos. Mientras que el modelado orientada a objetos especifica con claridad la interfaz de *inbound* de un objeto (el lado "tiene" de una interfaz), casi ninguna herramienta permite expresar con naturalidad la interfaz opuesta de *outbound*, popularmente conocida como "necesita". Sólo el diseño por contrato de Bertrand Meyer soporta este método de dos vías en su núcleo; en UML se debió implementar como extensión o metamodelo. UML tampoco considera los conectores como objetos de primera clase, por lo cual se deben implementar extensiones mediante ROOM o de alguna otra manera [Abd00].

(8) Klaus-Dieter Schewe, en particular, ha reseñado lo que él interpreta como un cúmulo de limitaciones y oscuridades de UML, popularizando además su consideración como un "dinosaurio moderno" [Sch00]. Al cabo de un análisis pormenorizado del que aquí no podemos dar cuenta sin excedernos de espacio, Schewe estima que, dando continuidad a una vieja polémica iniciada por E. F. Codd cuando éste cuestionó a los modelos en Entidad-Relación, UML es el ganador contemporáneo cuando se trata de falta de definiciones precisas, falta de una clara semántica, falta de claridad con respecto a los niveles de abstracción y falta de una metodología pragmática.

(9) Hoffmeister y otros [HNS99] alegan que UML es deficiente para describir correspondencias tales como el mapeo entre elementos en diferentes vistas, que serían mejor representadas en una tabla; faltan también en UML los elementos para modelar comunicación *peer-to-peer.* Los diagramas de secuencia de UML no son tampoco adecuados para soportar la configuración dinámica de un sistema, ni para describir secuencias generales de actividades.

A despecho que los responsables de UML insistan en la precisión y rigor de su semántica, se han organizado simposios y *workshops* enteros para analizar las limitaciones del modelo semántico de UML y su falta de recursos para detectar tempranamente errores de requerimiento y diseño [OOS99]. Si bien se reconoce que con UML es posible representar virtualmente cualquier cosa, incluso fenómenos y procesos que no son software, se debe admitir que muchas veces no existen formas *estándar* de materializar esas representaciones, de modo que no pueden intercambiarse modelos entre diversas herramientas y contextos sin pérdida de información. Se ha dicho que ni las clases, ni los componentes, ni los *packages*, ni los subsistemas de UML son unidades arquitectónicas adecuadas: las clases están tecnológicamente sesgadas hacia la OO y representan entidades de granularidad demasiado pequeña; los componentes "representan piezas físicas de implementación de un sistema" y por ser unidades de implementación y no de diseño, es evidente que existen en el nivel indebido de abstracción; un *package* carece de la estructura interna necesaria; los subsistemas pueden no aparecer en los diagramas de despliegue, o mezclarse con otras entidades de diferente granuralidad y propósito, carecen del concepto de puerto y su semántica y pragmática se han estimado caóticas [StöS/f].

Una debilidad más seria tiene que ver con la falta de modelos causales rigurosos; aunque UML proporciona herramientas para modelar requerimientos de comportamiento (diagramas de estado, de actividad y de secuencia), al menos en UML 1.x no existe diferencia alguna, por ejemplo, entre mensajes opcionales y mensajes requeridos; se pueden "colorear" los mensajes con restricciones, por cierto, pero es imposible hacerlo de una manera estándar. Asimismo, aunque los objetos de UML se pueden descomponer en piezas más pequeñas, no es posible hacer lo propio con los mensajes. También es palpable la sensación de que su soporte para componentes, subsistemas y servicios necesita ser mejorada sustancialmente (incluso en UML 2.0), que los modelos de implementación son inmaduros y que no hay una clara diferenciación o un orden claro de correspondencias entre modelos notacionales y meta-modelos, o entre análisis y diseño [Dou00] [AW99] [KroS/f].

Robert Allen [All97] ha señalado además que si bien se pueden diagramar comportamientos, los diagramas no distinguen entre un patrón de interacción y la conducta de los participantes en esa interacción. De este modo, es difícil razonar sobre un protocolo de interacción y la conformidad de un componente a ese protocolo, porque la interacción se define en términos de una determinada colección de componentes. Si los componentes coinciden exactamente con la conducta global de las máquinas de estados, entonces puede decirse que obdecen trivialmente el protocolo. Pero si no lo hacen, no hay forma claramente definida de separar las partes de la conducta del componente que son relevantes a una interacción en particular y analizarlas.

En una presentación poco conocida de uno de los líderes de la arquitectura de software, David Garlan, se señalan deficiencias de UML en relación con lo que podría ser la representación de estilos: los *profiles* de UML son demasiado densos y pesados para esa tarea, mientras que los *packages* proporcionan agregación, pero no restricciones. Por otra parte, las asociaciones de UML no sirven para denotar conectores, porque no pueden definirse independientemente del contexto y no es posible representar sub-estructuras. La información arquitectónica se termina construyendo de alguna manera, pero el resultado es un embrollo visual y conceptual; y aunque la simbología se puede visualizar con provecho y está bellamente lograda en imágenes, no hay nada más que hacer con ella que usarla a los efectos de la documentación [Gars/f].

Las limitaciones de UML en materia de *round-trip engineering* han sido documentadas con alguna frecuencia. En el meta-modelo de UML se definen numerosos conceptos que no aparecen en los modelos de implementación, que usualmente son lenguajes orientados a objeto; “Agregación”, “Asociación”, “Composición” y “Restricción” (*constraint*), “Estereotipo” son ejemplos de ello, pero hay muchos más. Herramientas de CASE y modelado industriales de alta calidad, como Rational Rose, TogetherSoft Together o JBuilder de Borland o productos de código abierto como ArgoUML, no poseen definiciones claras de las relaciones de clase binarias como asociación, agregación o composición; distinguen entre ellas a nivel gráfico, pero el código que se sintetiza para las diferentes clases binarias es el mismo. Las herramientas de reingeniería producen por lo tanto relaciones erróneas e inconsistentes; simplemente generando código a partir del diagrama y volviendo a generar el diagrama a partir del código se obtiene un modelo distinto [GAD+02]. De la misma manera, UML carece de conceptos de invocación y acceso; si bien las herramientas existentes resuelven de alguna manera estos problemas, en general lo hacen en forma idiosincrática y propietaria. En consecuencia, las herramientas no son interoperables. Las propuestas de solución existentes (por ejemplo, FAMIX) deben implementar meta-modelos que no son los de UML [DDT99].

Ya se ha dicho que UML no es en modo alguno un ADL en el sentido usual de la expresión. Los manuales contemporáneos de UML carecen del concepto de estilo y sus definiciones de “arquitectura” no guardan relación con lo que ella significa en el campo de los ADLs [BRJ99] [Lar03]. Sin embargo, debido al hecho de que constituye una herramienta de uso habitual en modelado, importantes autoridades en ADLs, siguiendo a Nenad Medvidovic, han investigado la posibilidad de utilizarlo como metalenguaje para implementar la semántica de al menos dos ADLs, que son C2 SADL y Wright [RMR+98]. También se ha afirmado que el meta-modelo de UML puede constituir un equivalente a la ontología arquitectónica de Acme.

Como quiera que sea, al no tipificar UML como un lenguaje homólogo a los ADLs que aquí se tratan no analizaremos sus rasgos y prestaciones en los términos que hasta aquí han sido habituales. Baste decir que, ya sea que se oponga o complemente al resto de las herramientas descriptivas, existen innumerables recursos alrededor de él para modelar sistemas en la plataforma de referencia. Con Visio Enterprise Architect de Microsoft se pueden crear, de hecho, los nueve tipos de diagramas de UML 1.*.

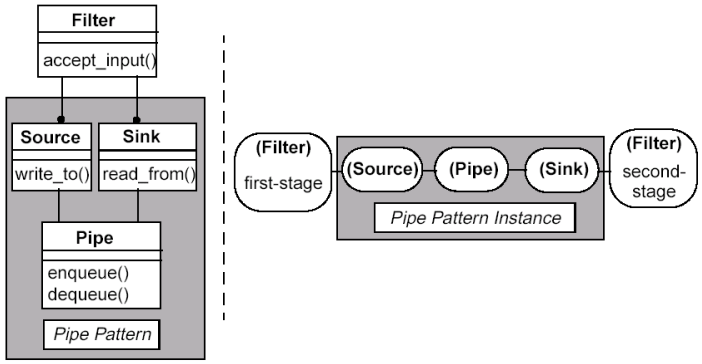


Fig. 7 – Tubería y filtro en UML

A fin de vincular el modelado basado en objetos con las problemáticas de ADLs y estilo que son propias de este estudio, nos ha parecido oportuno incluir una representación en notación OMT del modelo de referencia de tubería y filtros. En esta imagen, tomada de [MKM+96] se incorpora también la idea de concebir la tubería no sólo como una entidad de diseño de primer orden, sino como un patrón de diseño. Anticipamos de esta manera una cuestión que será tratada al cierre de este estudio, la cual concierne a la relación entre los ADLs y el campo de los patrones de diseño, en plena expansión a comienzos del siglo XXI.

UniCon

UniCon (Universal Connector Support) es un ADL desarrollado por Mary Shaw y otros [SDK+95]. Proporciona una herramienta de diseño para construir configuraciones ejecutables basadas en tipos de componentes, implementaciones y "conexiones expertas" que soportan tipos particulares de conectores. UniCon se asemeja a Darwin en la medida en que proporciona herramientas para desarrollar configuraciones ejecutables de caja negra y posee un número fijo de tipos de interacción, pero el modelo de conectores de ambos ADLs es distinto.

Oficialmente se define como un ADL cuyo foco apunta a soportar la variedad de partes y estilos que se encuentra en la vida real y en la construcción de sistemas a partir de sus descripciones arquitectónicas. UniCon es el ADL propio del proyecto Vitruvius, cuyo objetivo es elucidar un nivel de abstracción de modo tal que se pueda capturar, organizar y tornar disponible la experiencia colectiva exitosa de los arquitectos de software.

El siguiente ejemplo muestra un modelo completo de tubería y filtro en UniCon, definiendo un primer filtro para procesamiento primario y un segundo para eliminación de vocales:

```
COMPONENT System
       INTERFACE is
```

```
            TYPE Filter
            PLAYER input IS StreamIn
                  SIGNATURE (“line”)
                  PORTBINDING (stdin)
            END input
            PLAYER output IS StreamOut
                  SIGNATURE (“line”)
                  PORTBINDING (stdout)
            END output
            PLAYER error IS StreamOut
                  SIGNATURE (“line”)
                  PORTBINDING (stderr)
            END error
      END INTERFACE
      IMPLEMENTATION IS
            USES C INTERFACE CatFile
            USES R INTERFACE RemoveVowels
            USES P PROTOCOL Unix-pipe
            BIND input TO C.input
            BIND output TO R.output
            BIND C.error TO error
            BIND R.error TO error
            BIND P.err to error
            CONNECT C.output TO P.source
            CONNECT P.sink to R.input
      END IMPLEMENTATION
END System
COMPONENT CatFile
      INTERFACE is
            TYPE Filter - - como antes sin binding a puertos
      END INTERFACE
      IMPLEMENTATION IS
            VARIANT CatFile IN “catfile”
                  IMPLTYPE (Executable)
                  END CatFile
      END IMPLEMENTATION
END CatFile
COMPONENT RemoveVowels
      INTERFACE is
            TYPE Filter - - como antes sin binding a puertos
      END INTERFACE
      IMPLEMENTATION IS
            VARIANT RemoveVowels IN “remove”
            IMPLTYPE (Executable)
            END RemoveVowels
      END IMPLEMENTATION
END RemoveVowels
CONNECTOR Unix-pipe
      PROTOCOL IS
            TYPE Pipe
            ROLE source IS source
                  MAXCONNS (1)
                  END source
            ROLE sink IS sink
                  MAXCONNS (1)
                  END sink
            ROLE err IS sink
                  MAXCONNS (1)
                  END err
            END PROTOCOL
      IMPLEMENTATION IS
            BUILTIN
      END IMPLEMENTATION
```

```
END Unix-Pipe
```

Objetivo principal – El propósito de UniCon es generar código ejecutable a partir de una descripción, a partir de componentes primitivos adecuados. UniCon se destaca por su capacidad de manejo de métodos de análisis de tiempo real a través de RMA (Rate Monotonic Analysis).

Sitio de referencia - http://www-2.cs.cmu.edu/People/UniCon/ - El link de distribución de UniCon en la Universidad Carnegie Mellon estaba muerto o inaccesible en el momento de redacción de este documento (enero de 2004). El sitio de Vitrivius fue modificado por última vez en diciembre de 1998.

Estilos - UniCon no proporciona medios para describir o delinear familias de sistemas o estilos.

Interfaces – En UniCon los puntos de interfaces de los componentes se llaman *players*. Estos players poseen un tipo que indica la naturaleza de la interacción esperada, y un conjunto de propiedades que detalla la interacción del componente en relación con esa interfaz. En el momento de configuración, los *players* de los componentes se asocian con los roles de los conectores.

Semántica – UniCon sólo soporta cierta clase de información semántica en listas de propiedades.

Interfaz gráfica – UniCon soporta notación gráfica.

Generación de código – UniCon genera código C mediante el procedimiento de asociar elementos arquitectónicos a construcciones de implementación, que en este caso serían archivos que contienen código fuente. Sin embargo, habría algunos problemas con esta forma de implementación, dado que presuponer que esa asociación vaya a ser siempre uno-a-uno puede resultar poco razonable. Después de todo, se supone que los ADLs deben describir los sistemas a un nivel de abstracción más elevado que el que es propio del código fuente. Tampoco hay garantía que los módulos de código fuente implementen la conducta deseada o que se pueda seguir el rastro de futuros cambios en esos módulos hasta la arquitectura y viceversa.

Observaciones – UniCon (igual que Darwin) carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos predefinidos. Los tipos de componente son definidos por enumeración, no siendo posible definir sub-tipos y careciendo por lo tanto de capacidad de evolución. Una ejemplificación razonable del caso testigo del estilo de tubería y filtros, a pesar de la simplicidad inherente a su estructura y de estar aquí tratándose de lenguajes descriptivos que deberían referirse a un elevado nivel de abstracción involucraría varias páginas de código; por esta razón omito el ejemplo, remitiendo a la documentación del manual de referencia del lenguaje que se encuentra en http://www-2.cs.cmu.edu/People/UniCon/reference-manual/Reference_Manual_1.html.

Disponibilidad de plataforma – La distribución de UniCon no se encuentra actualmente activa.

Weaves

Propuesto alrededor de 1991 [GR91], Weaves soporta la especificación de arquitecturas de flujo de datos. En particular, se especializa en el procesamiento en tiempo real de grandes volúmenes de datos emitidos por satélites meteorológicos. Es un ADL acaso demasiado ligado a un dominio específico, sobre el cual no hay abundancia de documentación disponible.

Wright

Se puede caracterizar sucintamente como una herramienta de formalización de conexiones arquitectónicas. Ha sido desarrollado por la Escuela de Ciencias Informáticas de la Universidad Carnegie Mellon, como parte del proyecto mayor ABLE. Este proyecto a su vez se articula en dos iniciativas: la producción de una herramienta de diseño, que ha sido Aesop, y una especificación formal de descripción de arquitecturas, que es propiamente Wright.

Objetivo principal – Wright es probablemente la herramienta más acorde con criterios académicos de métodos formales. Su objetivo declarado es la integración de una metodología formal con una descripción arquitectónica y la aplicación de procesos formales tales como álgebras de proceso y refinamiento de procesos a una verificación automatizada de las propiedades de las arquitecturas de software.

Sitio de referencia – La página de ABLE de la Universidad Carnegie Mellon se encuentra en http://www-2.cs.cmu.edu/afs/cs/project/able/www/able.html. La del proyecto Wright está en http://www-2.cs.cmu.edu/afs/cs/project/able/www/wright/index.html.

Estilos - En Wright se introduce un vocabulario común declarando un conjunto de tipos de componentes y conectores y un conjunto de restricciones. Cada una de las restricciones declaradas por un estilo representa un predicado que debe ser satisfecho por cualquier configuración de la que se declare que es miembro del estilo. La notación para las restricciones se basa en el cálculo de predicados de primer orden. Las restricciones se pueden referir a los conjuntos de componentes, conectores y *attachments*, a las puertos y a las computaciones de un componente específico y a los roles y ligamentos (*glue*) de un conector particular. Es asimismo posible definir sub-estilos que heredan todas las restricciones de los estilos de los que derivan. No existe, sin embargo, una manera de verificar la conformidad de una configuración a un estilo canónico estándar.

Interfaces - En Wright (igual que en Acme y Aesop) los puntos de interfaz se llaman puertos (*ports*).

Semántica – Mientras muchos lenguajes de tipo ADL no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, Wright y Rapide permiten modelar la conducta de sus componentes. En Wright existe una sección especial de la especificación llamada `Computation` que describe la funcionalidad de los componentes.

Análisis y verificación automática – Al basarse en CSP como modelo semántico, Wright permite analizar los conectores para verificar que no haya *deadlocks*. Wright define verificaciones de consistencia que se aplican estáticamente, y no mediante simulación.

Esto lo hace más confiable que, por ejemplo, Rapide. También se puede especificar una forma de compatibilidad entre un puerto y un rol, de modo que se pueda cambiar el proceso de un puerto por el proceso de un rol sin que la interacción del conector detecte que algo ha sido modificado. Esto permite implementar relaciones de refinamiento entre procesos. En Wright existen además recaudos (basados en teoremas formales relacionados con CSP) que garantizan que un conector esté libre de *deadlocks* en todos los escenarios posibles. Esta garantía no se obtiene directamente sobre Wright, sino implementando un verificador de modelos comercial para CSP llamado FDR. Herramientas complementarias gene-ran datos de entrada para FDR a partir de una especificación Wright. Cualquier herramienta de análisis o técnica que pueda usarse para CSP se puede utilizar también para especificaciones en Wright.

Interfaz gráfica – Wright no proporciona notación gráfica en su implementación nativa.

Generación de código – Wright se considera como una notación de modelado auto-contenida y no genera (al menos en forma directa) ninguna clase de código ejecutable.

El siguiente ejemplo en código Wright correspondiente a nuestro caso canónico de tubería y filtros ilustra la sintaxis para el estilo correspondiente y la remisión al proceso de verificación (no incluido). Presento primero la notación básica de Wright, la más elemental de las disponibles:

```
Style DataFiles
Component DataFile1
       Port File = Action [] Exit
       where {
              Exit = close -> Tick
              Action = read -> Action [] write -> Action
       }
       Computation = ...
       Component DataFile2
              Port File = Action [] Exit
              where {
                     Exit = close -> Tick
                     Action = read -> File [] write -> File
              }
              Computation = ...
       Connector Pipe
              Role Writer = write -> Writer |~| close -> Tick
              Role Reader = DoRead |~| ExitOnly
              where {
                     DoRead = read -> Reader [] readEOF -> ExitOnly
                     ExitOnly = close -> Tick
              }
              Glue = ...
End Style
```

La siguiente pieza de código, más realista, incluye la simbología *hardcore* de Wright según Robert Allen y David Garlan [AG96]:

```
Style pipe-and-filter
```

```
Interface Type DataInput = (read Æ → (data?x → DataInput
              [] end-of-data → close → √))
              ⌈⌉ (close →√)
Interface Type DataOutput = write → DataOutput ⌈⌉ close →√
       Connector Pipe
              Role Source = DataOutput
              Role Sink = DataInput
              Glue = Buf<>
                     where
              Buf<> = Source.write?x → Buf<x> [] Source.close → Closed<>
              Buf<s<x>> = Source.write?y → Buf<y>s<x>>
                     [] Source.close → Closeds<x>
                     [] Sink.read → Sink.data!x → Bufs
                     [] Sink.close → Killed
              Closed<s<x>> = Sink.read → Sink.data!x → Closed<s>
                     [] Sink.close → √
                     Closed<> = Sink.read → Sink.end-of-data → Sink.close → √
                     Killed = Source.write → Killed [] Source.close → √
              Constraints
                     ∀ c : Connectors • Type(c) = Pipe
                     ∀ c : Components • Filter(c)
                     where
                     Filter(c:Component) = ∀ p : Ports(c) • Type(p) = DataInput
                     ∨ Type(p) = DataOutput
End Style
```

La imagen ilustra la representación gráfica correspondiente al estilo:

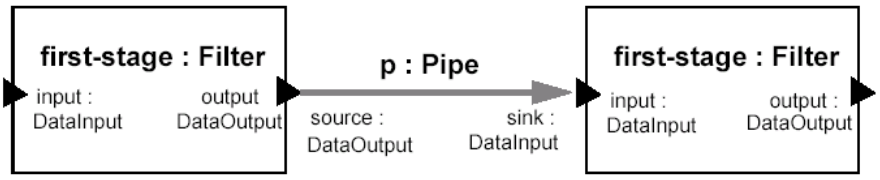


Fig. 8 – Diagrama correspondiente al código en Wright

Implementación de referencia – Aunque se ha señalado su falta de características explícitas de escalabilidad, Wright se utilizó para modelar y analizar la estructura de *runtime* del Departamento de Defensa de Estados Unidos. Se asegura que permitió condensar la especificación original y detectar varias inconsistencias en ella.

Disponibilidad de plataforma – Wright es en principio independiente de plataforma y es un lenguaje formal, antes que una herramienta paquetizada. Debido a que se basa en CSP (Communicating Sequential Process), cualquier solución que pueda tratar código CSP en plataforma Windows es apta para obtener ya sea una visualización del modelo o verificar la ausencia de deadlocks o la consistencia del modelo. El código susceptible de ser manejado por herramientas de CSP académicas o comerciales requiere tratamiento previo por un módulo de Wright que por el momento existe sólo para Linux o SunOS; pero una vez que se ha generado el CSP (lo cual puede hacerse manualmente insertando unos pocos *tags*) se puede tratar en Windows con cualquier solución ligada a ese lenguaje, ya sea FDR o alguna otra. CSP no está ligado a plataforma alguna, y fue elaborado a

mediados de la década de 1970 por Sir Tony Hoare, actualmente Senior Researcher en Microsoft Research, Cambridge.

Modelos computacionales y paradigmas de modelado

La descripción que se ha proporcionado hasta aquí no constituye un *scorecard* ni una evaluación sistemática de los diferentes ADLs, sino una revisión de los principales lenguajes descriptivos vigentes para que el lector arquitecto se arme de una idea más precisa de las opciones hoy en día disponibles si lo que se requiere una herramienta de descripción de arquitecturas.

Los arquitectos de software con mayor inclinación hacia las cuestiones teóricas y los métodos formales habrán podido apreciar la gran variedad de fundamentaciones semánticas y modelos computacionales que caracteriza al repertorio de los ADLs. La estructura de Darwin se basa en el cálculo Л; Wright es prácticamente una aplicación rigurosa del CSP de Hoare para su semántica y de lógica de primer orden para sus restricciones; LILEANNA se respalda en programación parametrizada e hiper-programación, así como en teoría categorial; los ADLs basados en eventos se fundan en Conjuntos de Eventos Parcialmente Ordenados (*Posets*). Casi todos los ADLs se acompañan de una gramática BNF explícita y algunos (como Armani) requieren también familiaridad con la construcción de un Arbol Sintáctico Abstracto (AST). Al menos un ADL contemporáneo, SAM, implementa redes de Petri de transición de predicados y lógica temporal de primer orden y tiempo lineal.

Un concepto importante que surge de haber considerado este abanico de posibilidades, es que no todo lo que tiene que ver con diseño requiere compulsivamente notaciones y modelos de objeto. Considerando los ADLs en su conjunto, habrá podido comprobarse que la tradición de modelado OO, como la que se plasma a partir de [RBP+91], juega en este campo un papel muy modesto, si es que juega alguno; la reciente tendencia a redefinir los ADLs en términos de mensajes, servicios, integración, XML Schemas, xADL, SOAP y sus derivaciones no hará más que consolidar ese estado de cosas.

En el campo del modelado arquitectónico, UML/OOP tiende a ser antagónico a la modalidad “estructural” representada por los ADLs. Grandes proyectos recientes, como el del equipo de JPL que está definiendo el *framework* de Mission Data Systems para la NASA, ha decidido hace pocos meses reemplazar UML por ADLs basados en XML [MMM03]. Correlativamente, como bien destaca Paul Clements en uno de sus *surveys*, muy pocos ADLs pueden manejar conceptos básicos en el modelado OO tales como herencia de clases y polimorfismo [Cle96]. De más está decir que ningún ADL a la fecha permite incorporar inflexiones más complejas y sutiles como programación orientada a aspectos, otro tópico caliente en lo que va del siglo [MPM+99]. En la generalidad de los casos, tampoco es relevante que así sea.

ADLs en ambientes Windows

En este estudio se ha podido comprobar que existe un amplio repertorio de implementaciones de ADLs y sus herramientas colaterales de última generación en plataforma Windows. Específicamente señalo Saage, un entorno para diseño arquitectó-

nico que utiliza el ADL de C2SADEL y que requiere Visual J++ (o Visual J#), COM y eventualmente Rational Rose (o DRADEL); también existen extensiones de Visio para C2, xADL y naturalmente UML. Visio para xADL se utiliza como la interfaz usuario de preferencia para *data binding* de las bibliotecas de ArchEdit para xADL. En relación con xADL también se puede utilizar en Windows ArchStudio para modelado en gran escala, o Ménage para mantenimiento y gestión de arquitecturas cambiantes en tiempo de ejecución.

Si el lenguaje de elección es Darwin se puede utilizar SAA. Si en cambio es MetaH, se dispone del MetaH Graphical Editor de Honeywell, implementado sobre DoME y nativo de Windows. En lo que se refiere a Acme, el proyecto Isis y Vanderbilt proporcionan GME (Metaprogrammable Graphical Model Editor), un ambiente de múltiples vistas que, al ser meta-programable, se puede configurar para cubrir una rica variedad de formalismos visuales de diferentes dominios. Una segunda opción para Acme podría ser VisEd, un visualizador arquitectónico y editor de propiedades de GA Tech; una tercera, AcmeStudio; una cuarta, Acme Powerpoint Editor de ISI, la cual implementa COM. Los diseños de Acme y Rapide se pueden poner a prueba con Aladdin, un analizador de dependencia del Departamento de Ciencia de la Computación de la Universidad de Colorado. Cualquier ADL que se base en alguna extensión de *scripting* Tcl/Tk se habrán de implementar con facilidad en Windows. El modelado y el análisis de una arquitectura con Jacal es nativa de Windows y las próximas versiones del ADL utilizarán Visio como front-end.

Algunas herramientas son nativas de Win32, otras corren en CygWin, Interix o algún Xserver, otras necesitan JRE. Mientras los ADLs de elección estén vivos y mantengan su impulso no faltarán recursos y complementos, a menos que se escoja una plataforma muy alejada de la corriente principal para ejecutar el software de modelado. No necesariamente habrá que lidiar entonces manualmente con notaciones textuales que quizá resulten tan complejas como las de los propios lenguajes de implementación. Si bien el trabajo del arquitecto discurre a un nivel de abstracción más elevado que la del operario que escribe código, en ninguna parte está escrito que vaya a ser más simple. Por el contrario, está claro que desde el punto de vista del modelado con ADLs, el arquitecto debe dominar no sólo una práctica sino también una teoría, o diversas teorías convergentes.

Pero los ambientes gráficos son sólo una parte de la cuestión, un capítulo que dudosamente merezca ocupar el foco de una estrategia arquitectónica o constituirse en su parámetro decisivo; poseen la misma importancia relativa y circunstancial que los nombres o los URLs de especificaciones que hoy pueden estar y mañana desaparecer sin dar explicaciones. El núcleo de la relevancia y conveniencia de los ADLs tiene que ver más bien con las diferentes cuestiones formales involucradas en los distintos escenarios y estilos, y eventualmente con la relación entre el modelado arquitectónico y el inmenso campo, en plena ebullición, de los patrones arquitectónicos en general y los patrones de diseño en particular.

ADLs y Patrones

A primera vista, daría la impresión que la estrategia arquitectónica de Microsoft reposa más en la idea de patrones y prácticas que en lenguajes de descripción de diseño; pero si

se examina con más cuidado el concepto de patrones se verá que dichos lenguajes engranan clara y armónicamente en un mismo contexto global. El concepto de patrón fue acuñado por Christopher Alexander entre 1977 y 1979. De allí en más fue enriqueciéndose en una multitud de sentidos, abarcando un conjunto de prácticas para capturar y comunicar la experiencia de los diseñadores expertos, documentar *best practices*, establecer formas recurrentes de solución de problemas comunes, etcétera.

Las clasificaciones usuales en el campo de los patrones reconocen habitualmente que hay diferentes clases de ellos; por lo común se hace referencia a patrones de análisis, organizacionales, procedurales y de diseño. Esta última categoría tiene que ver con recurrencias específicas de diseño de software y sistemas, formas relativamente fijas de interacción entre componentes, y técnicas relacionadas con familias y estilos. En este sentido, es evidente que los ADLs denotan una clara relación con los patrones de diseño, de los que puede decirse que son una de las formas posibles de notación. Como en el caso de las heurísticas y recetas de Acme, muchas veces esos patrones son explícitos y el ADL opera como una forma regular para expresarlos en la descripción arquitectónica.

La correspondencia entre patrones, ADLs y estilos, sin embargo, no está establecida de una vez y para siempre porque no existe ni una taxonomía uniforme, ni una especificación unívoca que defina taxativamente cuántas clases de patrones y cuántos niveles existen en ingeniería, arquitectura y diseño de software desde que se concibe conceptualmente un sistema hasta que se lo programa. En la percepción de Jørgen Thelin, quien a su vez se funda en la nomenclatura y tipificación de Martin Fowler, por ejemplo, distintos niveles en los proyectos de desarrollo se vincularían con otras tantas clases de patrones: el diseño con patrones de código, el análisis con los patrones de modelos y los estilos arquitectónicos con los patrones de arquitectura [The03]. Pero aunque cada escuela y cada autor organiza la articulación del campo a su manera y se sirve de denominaciones idiosincráticas, sin duda es legítimo plantear una relación entre patrones y lenguajes de descripción de arquitectura y tratar de esclarecerla.

Los ADLs mantienen asimismo una frontera difusa con lo que desde Alexander en adelante se llamaron lenguajes de patrones, que se definen como sistemas de patrones organizados en una estructura o plantilla que orienta su aplicación [Ale79] [AIS+77]. Ambas ideas se consolidaron en una nutrida bibliografía, en la cual la obra de la llamada "Banda de los Cuatro" ha alcanzado una fama legendaria [Gof95]. En algunas contadas ocasiones, los lenguajes de patrones fueron reformulados como ADLs y también viceversa [Shaw96]. Recientemente se ha propuesto un nuevo estilo arquitectónico "de nueva generación", ABAS (Attribute-Based Architectural Style) [KC99], que explícitamente se propone la convergencia entre la arquitectura de alto nivel expresada en los ADLs y en los estilos con el *expertise*, las *best practices,* los *building blocks* y los patrones decantados en la disciplina de diseño de aplicaciones.

Un estudio esencial de Mary Shaw y Paul Clements analiza la compleja influencia de la teoría y la práctica de los patrones sobre los ADLs [SC96]. Estos autores consideran que los ADLs han sido propios de la comunidad de arquitectos de software, mientras que los patrones de diseño y sus respectivos lenguajes han prosperado entre los diseñadores de software, particularmente entre los grupos más ligados a la orientación a objetos. Naturalmente, ambas comunidades se superponen en más de un respecto. En lo que

respecta a la relación entre arquitectura y diseño, las discusiones mantenidas en el seno de OOSPLA y en otros foros han consensuado que los diseñadores basados en patrones operan a niveles de abstracción más bajo que el de los arquitectos, pero por encima del propio de los programadores. Por otra parte, Buschmann [BMR+96] ha documentado patrones utilizados como estilos arquitectónicos regidos por ADLs. Shaw y Clements concluyen su análisis alegando que los ADLs pueden beneficiarse incorporando elementos de tipo análogo a los patrones en las secciones que se refieren a estilos, plantillas y reglas de diseño. A similares conclusiones llegan Robert T. Monroe, Drew Kompanek, Ralph Melton y David Garlan [MKM+96].

Al lado de los ADLs, existen otros recursos para abordar la cuestión de los patrones de diseño. La estrategia arquitectónica de Microsoft contempla niveles de abstracción que hoy en día se resuelven, por ejemplo, mediante Logidex para .Net de LogicLibrary, el cual se encuentra en http://www.logiclibrary.com/logidex.htm. Este es un *engine* que tipifica más como un *software development asset* (SDA) que como una entidad semejante a un ADL. Un *asset* de este tipo incluye componentes, servicios, frameworks, artefactos asociados con el ciclo de vida de un desarrollo de software, patrones de diseño y documentación de *best practices*. Este paquete en particular es más un producto para desarrolladores arquitectónicamente orientados que para arquitectos que prefieran situarse en una tesitura más abstracta y alejada del código. En futuras versiones de VisualStudio .Net se prevé la incorporación de otros recursos, algunos de ellos construidos en torno de la edición Architect y otros desarrollados por terceras partes. Los ADLs complementan a estos *assets* y herramientas, lo mismo que lo seguirán haciendo las soluciones de modelado relacionadas con la tradición de UML.

Conclusiones

Nuestra experiencia en conferencias de industria indica que cuando se presentan temas como los lenguajes de descripción arquitectónica y eventualmente los estilos de arquitectura ante audiencias de arquitectos, son muy pocos quienes han oído hablar de semejantes tópicos, y muchos menos aún quienes poseen alguna experiencia de modelado genuinamente arquitectónico. Por lo general tiende a confundirse este modelado abstracto con el diseño concreto de la solución a través de UML, o con la articulación de patrones. Este es un claro indicador del hecho de que la arquitectura de software no ha sabido comunicar su propia función mediadora entre el requerimiento por un lado y el diseño y la implementación por el otro. El presente estudio ha procurado describir herramientas disponibles para el desempeño de esa función. La idea no ha sido examinar productos concretos de modelado con ADL, sino referir problemáticas formales involucradas en esa clase de representación.

De más está decir que los ADLs son convenientes pero no han demostrado aún ser imprescindibles. Numerosas piezas de software, desde sistemas operativos a aplicaciones de misión crítica, se realizaron echando mano de recursos de diseño ad hoc, o de formalismos incapaces de soportar un *round trip* evolutivo, expresar un estilo, implementar patrones o generar código. Aunque algunos se perdieron en el camino o quedaron limitados a experiencias en dominios específicos, y aunque en la era de las arquitecturas orientadas a servicios se encuentran en estado de fluidez y transición, los

ADLs, nacidos hace más de una década, han llegado para quedarse. La demanda que los ha generado se ha instalado como un tópico permanente de la arquitectura de software, y por eso nos ha parecido útil elaborar este examen.

Referencias bibliográficas

[Abd00] Aynur Abdurazik. "Suitability of the UML as an Architecture Description Language with applications to testing". Reporte ISE-TR-00-01, George Mason University. Febrero de 2000.

[AG96] Robert Allen y David Garlan, "The Wright Architectural Description Language", *Technical Report*, Carnegie Mellon University. Verano de 1996.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Nueva York, Oxford University Press, 1979.

[All97] Robert Allen. "A formal approach to Software Architecture". *Technical Report*, CMU-CS-97-144, 1997.

[AW99] David Akehurst y Gill Waters. "UML deficiencies from the perspective of automatic performance model generation". *OOSPLA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations*, Denver, http://www.cs.kent.ac.uk/pubs/1999/899/content.pdf, Noviembre de 1999.

[AIS+77] Christopher Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King y S. Angel. *A Pattern Language*. Nueva York, Oxford University Press, 1977.

[BEJ+93] Pam Binns, Matt Englehart, Mike Jackson y Steve Vestal. "Domain-Specific Software Architectures for Guidance, Navigation, and Control". *Technical report*, Honeywell Technology Center, http://www-ast.tds-gn.lmco.com/arch/arch-ref.html, 1993.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stahl. *Pattern-Oriented Software Architecture: A System of Patterns*. Nueva York, Wiley, 1996.

[BRJ99] Grady Booch, James Rumbaugh e Ivar Jacobson. *El Lenguaje Unificado de Modelado*. Madrid, Addison-Wesley, 1999.

[Cle95] Paul C. Clements. "Formal Methods in Describing Architectures". En *Proceedings of the Workshop on Formal Methods and Architecture*, Monterey, California, 1995.

[DDT99] Serge Demeyer, Stéphane Ducasse y Sander Tichelaar. "Why FAMIX and not UML?: UML Shortcomings for coping with round-trop engineering". *UML'99 Conference Proceedings*, LNCS Series, Springer Verlag, 1999.

[DH01] Eric Dashofy y André van der Hoek. "Representing Product Family Architectures in an Extensible Architecture Description Language". En *Proceedings of the International Workshop on Product Family Engineering* (PFE-4), Bilbao, España, 2001.

[Dou00] Bruce Powel Douglas. "UML – The new language for real-timed embedded systems". http://wooddes.intranet.gr/papers/Douglass.pdf, 2000.

[Fie00] Roy Fielding. "Architectural styles and the design of network-based software architectures". Tesis de Doctorado. University of California at Irvine, 2000.

[Fux00] Ariel Fuxman. "A survey of architecture description languages". Febrero de 2000. Http://citeseer.nj.nec.com/fuxman00survey.html.

[GAD+02] Yann-Gaël Guéhéneuc, Hervé Albin-Amiot, Rémi Douence y Pierre Cointe. "Bridging the gap between modeling and programming languages". Reporte técnico, Object Technology International, Julio de 2002.

[GAO94] David Garlan, Robert Allen y John Ockerbloom. "Exploiting style in architectural design environments". En Proceedings of the SIGSOFT '94 Symposium on Foundations of Software Engineering. Nueva Orleans, diciembre de 1994.

[GarS/f] David Garlan. "Software architectures". Presentación en transparencias, http://www.sti.uniurb.it/events/sfm03sa/slides/garlan-B.ppt, sin fecha.

[GB99] Neil Goldman y Robert Balzer. "The ISI visual design editor". En *Proceedings of the 1999 IEEE International Conference on Visual Languages*, Setiembre de 1999.

[Gli00] Martin Glinz. "Problems and deficiencies of UML as a requirements specification language". En *Proceedings of the 10th International Workshop of Software Specification and Design* (IWSSD-10), pp. 11-22, San Diego, noviembre de 2000.

[GMW00] David Garlan, Robert Monroe y David Wile. "Acme: Architectural description of component-based systems". *Foundations of Component-Based Systems*, Gary T. Leavens y Murali Sitaraman (eds), Cambridge University Press, pp. 47-68, 2000.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GR91] Michael Gorlick y Rami Razouk. "Using Weaves for Software Construction and Analysis". En *13th International Conference on Software Engineering*, Austin, Mayo de 1991.

[HNS99] Christine Hofmeister, Robert Nord y Dilip Soni. "Describing Software Architecture with UML". En *Proceedings of the First Working IFIP Conference on Software Architecture*, IEEE Computer Society Press, pp. 145-160, San Antonio, Febrero de 1999.

[KC94] Paul Kogut y Paul Clements. "Features of Architecture Description Languages". Borrador de un CMU/SEI Technical Report, Diciembre de 1994.

[KC95] Paul Kogut y Paul Clements. “Feature Analysis of Architecture Description Languages”. En *Proceedings of the Software Technology Conference (STC’95)*, Salt Lake City, Abril de 1995.

[KK99] Mark Klein y Rick Kazman. “Attribute-Based Architectural Styles”, *Technical Report, CMU/SEI-99-TR-022, ESC-TR-99-022*, Octubre de 1999.

[KM95] Paul Kogut y Robert May. “Features of Architecture Description Languages”. *Software Technology Conference*, Salt Lake City, Abril de 1995.

[KroS/f] John Krogstie. “UML, a good basis for the development of models of high quality?”. Andersen Consulting Noruega & IDI, NTNU, http://dataforeningen.no/ostlandet/metoder/krogstie.pdf, sin fecha.

[Lar03] Craig Larman. *UML y Patrones*. 2ª edición, Madrid, Prentice Hall, 2003.

[LV95] David Luckham y James Vera. “An Event-Based Architecture Definition Language”. *IEEE Transactions on Software Engineering*, pp. 717-734, Setiembre de 1995.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach y Jeff Kramer. “Specifying distributed software architectures”. En *Proceedings of the Fifth European Software Engineering Conference, ESEC’95*, Setiembre de 1995.

[Med96] Neno Medvidovic. “A classification and comparison framework for software Architecture Description Languages”. Technical Report UCI-ICS-97-02, 1996.

[MK96] Jeff Magee y Jeff Kramer. “Dynamic structure in software architectures”. En *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 3–14, San Fransisco, Octubre de 1996.

[MKM+96] Robert Monroe, Drew Kompanek, Ralph Melton, David Garlan. “Stylized architecture, design patterns, and objects”. *Research Report*. Setiembre de 1996.

[MMM03] Nenad Medvidovic, Sam Malek, Marija Mikic-Rakic. “Software architectures and embedded systems”. Presentación en *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*. Chicago, Illinois, 24 al 26 de Setiembre de 2003.

[Mon98] Robert Monroe. “Capturing software architecture design expertise with Armani”. *Technical Report CMU-CS-163*, Carnegie Mellon University, Octubre de 1998.

[MPM+99] A. Navasa, M. A. Pérez, J. M. Murillo y J. Hernández. “Aspect oriented software architecture: A structural perspective”. Reporte del proyecto CICYT, TIC 99-1083-C2-02, 1999.

[OOS99] OOSPLA’99, Workshop #23: “Rigorous modeling and analysis with UML: Challenges and Limitations”. Denver, Noviembre de 1999.

[Par76] David Parnas. “On the Design and Development of Program Families.” *IEEE Transactions on Software Engineering* SE-2, pp. 1-9, Marzo de 1976.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy y William Lorensen. *Object-oriented modeling and design*. Englewood Cliffs, Prentice-Hall, 1991.

[Rey04] Carlos Reynoso. "Estilos y patrones en la estrategia de arquitectura de Microsoft". [Ref**], Marzo de 2004.

[RMR+98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles y David S. Rosenblum. "Integrating Architecture Description Languages with a Standard Design Method." En *Proceedings of the 20th International Conference on Software Engineering* (ICSE'98), pp. 209-218, Kyoto, Japón, 19 al 25 de Abril de 1998.

[SC96] Mary Shaw y Paul Clements. "How should patterns influence Architecture Description Languages?". *Working Paper for DARPA EDCS Community*, 1996.

[Sch00] Klaus-Dieter Schewe. "UML: A modern dinosaur? – A critical analysis of the Unified Modelling Language". En H. Kangassalo, H. Jaakkola y E. Kawaguchi (eds.), *Proceedings of the 10th European-Japanese Conference on Information Modelling and Knowledge Bases*, Saariselk, Finlandia, 2000.

[SDK+95] Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young y Gregory Zelesnik. "Abstractions for Software Architecture and Tools to Support Them". *IEEE Transactions on Software Engineering*, pp. 314-335, Abril de 1995.

[SG94] Mary Shaw y David Garlan. "Characteristics of Higher-Level Languages for Software Architecture". *Technical Report CMU-CS-94-210*, Carnegie Mellon University, Diciembre de 1994.

[Shaw94] Mary Shaw. "Patterns for software architecture". *First Annual Conference on the Pattern Languages of Programming*, 1994.

[Shaw96] Mary Shaw. "Some patterns for software architecture". Vlissides, Coplien & Kerth (eds.), *Pattern Languages of Program Design, Vol. 2*, pp. 255-269, Addison-Wesley, 1996.

[SG95] Mary Shaw y David Garlan. "Formulations and Formalisms in Software Architecture". Springer-Verlag, *Lecture Notes in Computer Science*, Volumen 1000, 1995.

[StöS/f] Harald Störrle. "Turning UML subsystems into architectural units". Reporte, Ludwig-Maximilians-Universität München, http://www.pst.informatik.uni-muenchen.de/personen/stoerrle/Veroeffentlichungen/PosPaperICSEFormat.pdf, sin fecha.

[Tem01] Theodor Tempelmeier. "Comments on Design Patterns for Embedded and Real-Time Systems". En: A. Schürr (ed.): *OMER-2 ("Object-Oriented Modeling of Embedded Realtime systems") Workshop Proceedings*. Mayo 9-12, 2001, Herrsching, Alemania. Bericht Nr. 2001-03, Universität der Bundeswehr München, Fakultät für Informatik, Mayo de 2001.

[Tem99] Theodor Tempelmeier. “UML is great for Embedded Systems – Isn’t it?” En: P. Hofmann, A. Schürr (eds.): *OMER (“Object-Oriented Modeling of Embedded Realtime systems”) Workshop Proceedings*. Mayo 28-29, 1999, Herrsching (Ammersee), Alemania. Bericht Nr. 1999-01, Universität der Bundeswehr München, Fakultät für Informatik, Mayo de 1999.

[The03] Jørgen Thelin. “A comparison of service-oriented, resource-oriented, and object-oriented architecture styles”. Presentación de Cape Clear Software, 2003.

[TMA+S/f] Richard Taylor, Nenad Medvidovic, Kennet Anderson, James Whitehead Jr, Jason Robbins, Kari Nies, Peyman Oreizy y Deborah Dubrow. “A component- and message-based architectural style for GUI software”. Reporte para el proyecto F30602-94-C-0218, Advanced Research Projects Agency, sin fecha.

[Ves93] Steve Vestal. “A cursory overview and comparison of four Architecture Description Languages”. *Technical Report*, Honeywell Technology Center, Febrero de 1993.

[Wolf97] Alexander Wolf. “Succeedings of the Second International Software Architecture Workshop” (ISAW-2). *ACM SIGSOFT Software Engineering Notes*, pp. 42-56, enero de 1997.