Revista Facultad de Ingeniería
ISSN: 0717-1072
facing@uta.cl
Universidad de Tarapacá
Chile

Bastarrica, María Cecilia; Gómez, David; Wilckens, Cristian
Input/Output Autómatas como lenguaje de definición de arquitecturas
Revista Facultad de Ingeniería, vol. 13, núm. 1, 2005, pp. 77-87
Universidad de Tarapacá
Arica, Chile

Rev. Fac. Ing. - Univ. Tarapacá, vol. 13 no. 1, 2005, pp. 77-87

INPUT/OUTPUT AUTÓMATAS COMO LENGUAJE DE DEFINICIÓN DE ARQUITECTURAS

María Cecilia Bastarrica[1] David Gómez[1] Cristian Wilckens[1]

Recibido el 14 de enero de 2004, aceptado el 25 de octubre de 2004

RESUMEN

Un lenguaje de descripción de arquitecturas (ADL) debe ser capaz de modelar componentes, conectores y configuraciones de software con una serie de características. Los Input/Output Autómatas (IOA) no fueron creados como un ADL sino como un lenguaje para especificación de sistemas concurrentes asincrónicos. Este artículo muestra cómo IOA puede también ser usado como ADL con múltiples ventajas.

Palabras clave: Lenguajes de descripción de arquitecturas, input/output, autómatas, arquitectura de software.

ABSTRACT

An architectural description language (ADL) must be able to model software components, connectors and configurations with a series of well defined characteristics. Input/Output Automata (IOA) were not created as an ADL but as a specification language for asinchronoust concurrent systems. In this paper we show how IOA can be used as an ADL with multiple advantages.

Keywords: Architecture description languages, input/output automata, software architecture.

INTRODUCCIÓN

Especificar la arquitectura de un software en una etapa temprana de su ciclo de vida es ventajoso según la mayor parte de las metodologías modernas de desarrollo de software. A pesar de esto, no existe actualmente una forma estándar de hacer la definición de una arquitectura. Los ADLs (Architecture Description Languages) son una solución tecnológica que ha sido propuesta desde la academia como una forma de notación para especificar formalmente arquitecturas de software [6] y que han sido aplicados en la industria con variados resultados.

La gran ventaja de usar ADLs sobre otras notaciones informales es que permiten una mejor comunicación entre el diseñador y los implementadores y lectores debido a que la formalidad no deja lugar a la ambigüedad, y además permite el análisis formal y temprano de las decisiones de diseño.

Existen muchos y diversos ADLs, unos con características particulares para cierto dominio de aplicación [7], [16] y otros de tipo general. Las ventajas de un cierto ADL estarán dadas entonces por su poder expresivo para especificar básicamente estructura y comportamiento, pero también sus formas de uso, la funcionalidad, el rendimiento, la flexibilidad, la capacidad de reutilización, la facilidad de comprensión y las restricciones económicas, tecnológicas y estéticas [12]. En [15] se publica un compendio de los ADLs existentes hasta el momento y se hace una caracterización de los elementos que sería deseable que tuviese un ADL.

Los Input/Output Autómatas (IOA) [13], [14] fueron creados como un lenguaje de especificación algebraica para describir sistemas concurrentes asincrónicos. Los sistemas se especifican como una composición de autómatas interactuantes, donde cada autómata puede tomar decisiones acerca de los mensajes que envía a su entorno y reacciona cada vez que recibe del entorno un mensaje para el cual está programado para reaccionar.

En este artículo mostramos cómo los IOA pueden ser usados para especificar componentes, conectores y configuraciones de los distintos patrones de arquitectura definidos en la literatura y podemos concluir que cumplen

[1] {cecilia,dagomez,cwilcken}@dcc.uchile.cl, Departamento de Ciencias de la Computación, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile. Blanco Encalada 2120, Santiago, Chile.

con todas y cada una de las características requeridas de un lenguaje de especificación para ser considerado un ADL según [15], lo cual lo pone en una buena posición para ser más usado en este ámbito.

En la siguiente sección damos una breve descripción de los Input/Output Autómatas. En la sección sobre Lenguajes de definición de arquitectura enumeramos las características que, según la literatura, debe tener un lenguaje de especificación para ser considerado un ADL. Distintos patrones de arquitectura se ejemplifican en a continuación y se analiza cómo en estas especificaciones IOA muestran las facetas requeridas de un ADL. Finalmente se incluyen las conclusiones.

INPUT/OUTPUT AUTÓMATAS

Los Input/Output Autómatas son un sistema de transiciones etiquetadas creado para modelar sistemas reactivos. Un autómata consta de un conjunto de acciones π que se clasifican como de *input*, de *output* o *internal*, un conjunto de estados *s* que incluye el conjunto no vacío de estados iniciales, un conjunto de *transiciones* de la forma $(s, π, s')$ que especifica los efectos de las acciones del autómata y un conjunto de *tasks* que son el conjunto de acciones controlables (internas o de output). La operación de un autómata se describe por su *ejecución* $s_0, \pi_1, s_1, ...$, la cual es una secuencia alternada de estados y acciones, y sus *trazas* están formadas por su comportamiento externamente visible, i.e. secuencias de acciones de input y output, que forman parte de las ejecuciones. Un autómata se dice que *implementa* a otro si sus trazas son también trazas del otro. Los I/O autómatas admiten el operador de *composición paralela*, el cual permite que, dado un conjunto de autómatas, una acción de output de un autómata se identifique con la acción de input de otro autómata y cada vez que una de ellas se ejecuta, también se ejecutan todas las que tienen el mismo nombre; este operador respeta la semántica de las trazas [7].

```
automaton canal (i,j : I, I, M : type)
    signature
        input send (m : M, const i, const j)
        output receive (m : M, const i, const j)
    states
        cola : Seq[M] := {}
    transitions
        input send (m, i, j)
                eff cola := cola _ m
        output receive (m, i, j)
                pre cola  {} ∧ m = head (cola)
              eff cola := tail (cola)
```

Fig. 1 Especificación IOA de un canal.

La Fig. 1 muestra un ejemplo de un autómata *canal* que puede combinarse con autómatas de tipo *proceso* para que los procesos se comuniquen a través del canal. Este ejemplo está publicado en [8]. Nótese que el canal es paramétrico en el tipo de datos que transmite (M), y también en el tipo de identificador que se usa para los procesos comunicantes (I).

La línea *signature* especifica las acciones que el autómata puede realizar, así como sus parámetros y sus respectivos tipos. Luego *states* especifica el conjunto de variables internas que definen el estado del autómata. Finalmente *transitions* especifica las precondiciones y los efectos de ejecutar cada acción. Nótese que las acciones de input no tienen precondición ya que el autómata no tiene control sobre los eventos producidos desde el exterior.

El lenguaje IOA está diseñado para permitir la descripción precisa y directa de los autómatas. Dado que el modelo de I/O autómatas es un modelo de sistemas reactivos más que un modelo de programas secuenciales, el lenguaje de descripción IOA refleja esta característica. Es así que no es un lenguaje de programación secuencial estándar con algunos constructos para modelar concurrencia e interacción, sino que estas facetas son parte esencial del lenguaje.

IOA fue diseñado para permitir tanto hacer demostraciones de corrección como para generar código. Esto hace que el diseño refleje esta tensión debido a que las características que hacen que un lenguaje sea apropiado para hacer demostraciones, e.g. estilo declarativo, simplicidad y soporte para no determinismo, son distintas de aquellas requeridas para permitir la generación de código, e.g. estilo imperativo, poder expresivo y determinismo.

Como parte de las decisiones de diseño tenemos:

- Los tipos de datos usados se definen axiomáticamente; esto facilita usar las especificaciones IOA como entrada de demostradores de teoremas. Existe una serie de definiciones de tipos que se suponen preexistentes y se da al usuario la posibilidad de especificar nuevos tipos usando Larch Shared Language (LSL) [10].

- Las definiciones de las transiciones pueden parametrizarse y los valores de estos parámetros pueden restringirse con predicados del tipo "where"'. Las transiciones pueden tener además parámetros del tipo "choose"', que no forman parte del nombre de la acción, pero que permiten que los parámetros se elijan de modo que cumplan con la precondición y luego pueden ser usados como parte de los efectos de la transición.

- Dado que ni el estilo declarativo ni el imperativo son completamente apropiados desde todo punto de vista para describir las transiciones, se permite usar los dos estilos, ya sea en forma conjunta o separada. Así los efectos de una transición pueden describirse como un programa, como un predicado o mediante una combinación de ellos, e.g. un programa que incluya elecciones no determinísticas explícitas, seguido de un predicado que restrinja estas elecciones.

- Las descripciones imperativas son lo más simples posible, y generalmente consisten de asignaciones (posiblemente no determinísticas), condiciones y simples ciclos acotados. Esta simplicidad tiene sentido debido a que las transiciones se supone que son ejecutadas atómicamente.

- Las variables pueden ser inicializadas mediante asignaciones corrientes o no determinísticas. El estado inicial completo puede también restringirse mediante un predicado.

- La definición de los autómatas completos puede ser parametrizada.

- Existe una notación explícita para la composición paralela. Para describir los valores de las variables de estado de un autómata compuesto, se usa la convención de usar el nombre del autómata del cual es parte como prefijo del nombre de cada variable de estado. Se pueden abreviar algunos de estos nombres cuando no existe ambigüedad.

- Existen notaciones explícitas para ocultar algunas acciones de output, para afirmar que un predicado es un invariante del autómata, o que existe una relación binaria de simulación (un autómata implementa a otro) entre un autómata y otro.

LENGUAJES DE DEFINICIÓN DE ARQUITECTURAS

Un lenguaje de definición de arquitecturas (ADL) sirve para describir una arquitectura de software. Según Garlan y Shaw [16], arquitectura de software es:

> "…la descripción de los elementos de los cuales un sistema está compuesto, sus interacciones, patrones que guían su composición y las restricciones a estos patrones."

Esto implica el conjunto de decisiones significativas respecto de la organización de un sistema de software. Estas decisiones incluyen seleccionar los elementos estructurales y las interfaces mediante las cuales se conectan, la organización de estos elementos y la topología de las conexiones, así como el comportamiento de estos elementos. La definición de la arquitectura usando un ADL incluye la posibilidad de especificar estructura y comportamiento, pero también es importante su forma de uso, funcionalidad, flexibilidad, rendimiento, restricciones y facilidad de comprensión. Un ADL también deberá ser capaz de facilitar el modelamiento de sistemas de software que sigan distintos patrones de arquitectura [4].

Un lenguaje de definición de arquitecturas debe hacerse cargo de todas estas características. Según Medvidovic et al. [14], existen ciertas características que son deseables en un ADL; éstas se resumen en la Tabla 1 y se detallan en las siguientes secciones. Los elementos indicados en negritas (componentes, conectores, configuraciones e interfaces de componentes) son los mínimos necesarios para que un lenguaje de especificación pueda ser considerado un ADL.

Tabla 1 Características de modelamiento requeridas para un ADL.

Elemento	**Características**
Componentes	Interfaces Tipos Semántica Evolución Propiedades no funcionales
Conectores	Interfaces Tipos Semántica Evolución Propiedades no funcionales
Configuraciones	Comprensibilidad Composición jerárquica Refinamiento y seguimiento Heterogeneidad Escalabilidad Evolucionabilidad Dinamismo Restricciones Propiedades no funcionales

Para realizar la descripción de la arquitectura los ADLs utilizan esencialmente componentes que son las piezas físicas de la implementación de un sistema, y los conectores para modelar la interacción. Estos conectores pueden ser implícitos en algunos ADLs o bien modelarse como elementos de primera clase [1], dependiendo del ADL y del patrón de arquitectura que se esté modelando.

Componentes

Las componentes son las entidades computacionales activas de un sistema. Ellas realizan tareas mediante cómputo interno y comunicación externa con el resto del sistema. La relación entre una componente y su entorno se define explícitamente como una colección de puntos de interacción o puertos. Los puertos se pueden generalizar como la noción de una interfaz de un módulo. En una forma simplificada, un puerto puede representar un procedimiento que puede ser llamado por otra componente, pero también puede ser un conjunto complejo de eventos que se disparan simultáneamente. Las componentes pueden ser pequeñas como un procedimiento o grandes como una aplicación entera encapsulada.

Una componente tiene sus propios datos y espacio de ejecución independientes, aunque podría compartirlos con otras componentes. Las características de una componente que un ADL debe ser capaz de modelar son las siguientes:

Interfaces. Es el conjunto de puntos de interacción entre una componente y su entorno. La interfaz especifica los servicios que provee una componente (operaciones, mensajes y variables) y también los servicios que la componente requiere de otras componentes del sistema. Es deseable que los ADLs permitan definir los tipos de las interfaces de las componentes de modo de maximizar la flexibilidad y la reutilización de sus definiciones.

Tipos. Los tipos de una componente son abstracciones que encapsulan funcionalidades en bloques reutilizables. Un tipo de componente puede ser instanciado múltiples veces en una sola arquitectura o ser reutilizado en diferentes arquitecturas. Los tipos de componentes pueden ser parametrizados facilitando así aún más su reutilización. El modelamiento explícito de tipos ayuda a analizar y entender una arquitectura en la que los tipos son compartidos por todas sus instancias.

Semántica. Se define la semántica como un modelo de alto nivel del comportamiento de la componente. Este modelo es necesario para realizar análisis, reforzar restricciones arquitecturales, y para asegurar la consistencia entre diferentes niveles de abstracción. Los modelos semánticos pueden ir desde expresar la información semántica como una lista de propiedades de la componente hasta modelos de comportamiento dinámico.

Restricciones. Las restricciones son propiedades o afirmaciones sobre un sistema o alguna de sus partes. Una componente puede ser restringida mediante atributos, e.g. restringiendo el número de asociaciones que puede tener un puerto, o indicando atributos no funcionales, e.g. tiempo de ejecución o deadlines.

Evolución. Los ADLs pueden permitir la evolución de componentes definiendo tipos de componentes y permitiendo el refinamiento mediante subtipado (subtyping).

Propiedades no funcionales. Estas propiedades no tienen relación con la funcionalidad sino con características de calidad de la componente tales como seguridad, rendimiento y portabilidad.

La mayoría de los ADLs tiene a las componentes como sus principales elementos de especificación; todos modelan interfaces y distinguen entre tipos e instancias de componentes. Por otro lado, casi ningún ADL soporta evolución ni especificación de propiedades no funcionales [14].

Conectores

Los conectores definen la interacción entre los componentes. Cada conector provee de una forma para que una colección de puertos esté en contacto y define lógicamente el protocolo a través del cual un conjunto de componentes puede interactuar. Los puertos definen los puntos de interacción de los conectores. Al igual que las componentes, un conector tiene una interfaz, la cual consiste en un conjunto de roles. Cada rol define el comportamiento esperado de uno de los participantes en la interacción. El comportamiento total de un conector está definido por un protocolo. Por lo tanto las características principales que debe tener un conector según [14] son las siguientes:

Interfaz. Es el conjunto de puntos de interacción entre un conector y una componente u otro conector. Las interfaces de conectores permiten una conectividad apropiada entre componentes y su interacción en una arquitectura. En general, cuando un ADL soporta conectores como entidades de primera clase [1], entonces soporta explícitamente también la especificación de las interfaces de estos conectores.

Tipos. Tipos de conectores son abstracciones que encapsulan la comunicación, coordinación y decisiones de mediación de componentes. Interacciones de nivel de arquitecturas son caracterizados por protocolos complejos identificados para diferentes patrones de arquitectura [4].

Semántica. La semántica de conectores se define como un modelo de alto nivel del comportamiento de un conector. La semántica expresa funcionalidad a nivel de la aplicación y la especificación de protocolos de interacción.

Restricciones. Las restricciones de los conectores aseguran la adherencia a protocolos de interacción. Además establecen dependencias entre conectores y refuerzan el uso de límites.

Evolución. La evolución de un conector se refiere a la modificación de las propiedades de un conector, o sea, de la modificación de su interfaz, semántica, o restricciones entre las dos.

Propiedades no funcionales. Las propiedades no funcionales de los conectores no se pueden derivar completamente de la especificación de su semántica. Representan requerimientos para implementar correctamente conectores. Permiten simulaciones de comportamiento en tiempo de ejecución, reforzamiento de restricciones, etc.

Distintos ADLs modelan conectores de variadas formas y con variados nombres. Muchos modelan conectores explícitamente y los llaman conectores, y otros como servicios de transporte.

Configuraciones

Una configuración o topología es una colección de instancias de componentes que interactúan mediante instancias de conectores. En otras palabras, una topología es un grafo de componentes y conectores conectados que describen la estructura de la arquitectura. Las características de nivel de configuración se agrupan en tres categorías generales:

- calidad de la descripción de la configuración: entendimiento, composición, refinamiento y seguimiento, heterogeneidad.
- calidad de la descripción del sistema: heterogeneidad, escalabilidad, evolución y dinamismo.
- propiedades de la descripción del sistema: dinamismo, restricciones y propiedades no funcionales.

Especificaciones entendibles. Uno de los roles de la arquitectura de software es el de servir como conducto de comunicación y así facilitar el entendimiento del sistema y la abstracción de alto nivel. Por lo tanto los ADLs deben ser capaces de modelar información estructural con una sintaxis comprensible.

Composición jerárquica. Es un mecanismo que permite que una arquitectura sea definida con distintos niveles de detalle. Estructuras y comportamiento complejos pueden ser representados explícitamente o abstractamente como una composición de componentes y conectores simples.

Refinamiento y seguimiento. Los ADLs deben proveer refinamiento de arquitecturas en sistemas ejecutables y seguimiento de los cambios a través de los distintos niveles.

Heterogeneidad. Las arquitecturas de software deben facilitar el desarrollo de sistemas de gran escala mediante la existencia de componentes y conectores de distinta granularidad, posiblemente especificadas en diferentes lenguajes de modelamiento e implementadas en diferentes lenguajes de programación.

Escalabilidad. Los ADLs deben soportar la especificación y el desarrollo de sistemas de gran escala que pueden crecer en el futuro.

Evolucionabilidad. Una arquitectura evoluciona para reflejar y permitir la evolución de familias de sistemas de software.

Dinamismo. Se refiere a modificar la arquitectura y reflejar esas modificaciones en el sistema mientras éste se ejecuta. Es importante el soporte de dinamismo en sistema como control de tráfico, en donde la disponibilidad y seguridad son críticas.

Restricciones. Restricciones que representan dependencias en una configuración complementan aquellas específicas a las componentes y conectores.

Propiedades no funcionales. Algunas de las propiedades no funcionales están a nivel del sistema más que a nivel de componentes y conectores individuales. Propiedades no funcionales a nivel de sistema son necesarias para seleccionar apropiadamente componentes y conectores, realizar análisis y reforzar restricciones, entre otras.

Otras características deseables

Según Medvidovic et al. [14], otra de las características más deseables de un ADL es que existan herramientas asociadas de apoyo tanto para hacer análisis como para hacer simulación de la ejecución de las arquitecturas especificadas. Si bien esta característica no es intrínseca

al ADL, es algo a tener en cuenta a la hora de hacer una elección ya que esto afecta enormemente su usabilidad y la robustez de las especificaciones puede asegurarse en una etapa más temprana.

IOA COMO ADL

Como forma de mostrar que IOA puede ser considerado como un ADL, mostramos en esta Sección cómo diversos patrones de arquitectura tales como tubos y filtros, memoria compartida o invocación remota pueden ser especificados de forma natural u\-san\-do IOA. Estos patrones pueden encontrarse especificados en la literatura usando otras notaciones tales como Wright [1] o Z [1].

Patrones de arquitectura en IOA

En IOA, tanto las componentes como los conectores se especifican como autómatas. Es así que en una arquitectura de memoria compartida tanto los datos de la memoria que es compartida (conector) como las componentes que hacen uso de ella están representados por autómatas. En forma similar, los tubos y los filtros son también autómatas que pueden combinarse en una composición paralela. Finalmente, en la invocación remota, típico tipo de comunicación que se da en arquitecturas cliente-servidor, tanto cliente como servidor son componentes modelados como autómatas, pero su comunicación se da a través de un modelado particular de la interfaz del cliente que produce que éste se bloquee hasta recibir la respuesta a la invocación remota.

```
automaton Repositorio (Info : type)
    signature
        input colocar (dato: Info)
        output sacar (dato: Info)
    states
        dato_actual: Info
    transitions
        input colocar (dato)
            eff dato_actual := dato
        output sacar (dato)
                    pre dato_actual = dato
```

Fig. 2 Especificación de un repositorio.

Memoria Compartida. Una primera aproximación a este conector consiste en tener un único repositorio, en el cual todas las componentes pueden eventualmente colocar información, y sacar la información previamente guardada por otras componentes. Los datos guardados son de tipo Info, el cual puede ser un tipo simple o compuesto.

Un segundo paso sería permitir que en una arquitectura haya varios repositorios como el ya construido, y que funcionen de modo independiente unos de otros. Esto se logra añadiendo a cada repositorio un identificador único (el cual será de tipo Id), tal como lo muestra la Fig. 3. Cada instancia de este tipo genérico de autómata se identificará con un valor único de su identificador.

```
automaton RepositorioGenerico (Info : type, ident : Id)
    signature
        input colocar (dato: Info, const ident: Id)
        output sacar (dato: Info, const ident: Id)
    states
        dato_actual: Info
    transitions
        input colocar (dato, ident)
            eff dato_actual := dato
        output sacar (dato, ident)
            pre dato_actual = dato
```

Fig. 3 Especificación de un repositorio múltiple.

Incluye una acción de entrada getInfo, que indica al repositorio que ha recibido una solicitud de envío de información; como consecuencia, se habilita la operación *sacar* que informa al entorno del valor de *dato_actual*.

En ninguno de los tres autómatas hemos definido un valor inicial para las variables de estado, así que éstas tendrán inicialmente valores arbitrarios, los cuales eventualmente pueden ser leídos por las componentes que así lo requieran.

```
automaton RepositorioPasivo (Info : type)
    signature
        input colocar (dato: Info),
            getInfo
        output sacar (dato: Info)
    states
        dato_actual: Info,
        solicitud: Bool := false
    transitions
        input colocar (dato)
            eff dato_actual := dato
        input getInfo
            eff solicitud := true
        output sacar (dato)
            pre solicitud ∧ dato_actual = dato
            eff solicitud := false
```

Fig. 4 Especificación de un repositorio pasivo.

Tubos y filtros. La Fig. 5 muestra un tubo simple que maneja una cola de mensajes de tipo *Info*.

```
automaton Tubo (Info : type)
    signature
        input enviar (m : Info)
        output recibir (m : Info)
    states
        mensajes: Seq[Info] := {}
    transitions
        input enviar (m)
            eff mensajes := mensajes ⊦ m
        output recibir (m)
            pre mensajes ~= {} ∧
                m = head (mensajes)
            eff mensajes := tail (mensajes)
```

Fig. 5 Especificación de un tubo simple.

Si se desea contar con varios tubos, basta agregar un identificador, al igual que para el autómata repositorio múltiple de memoria compartida.

Notar que en este caso sí hemos restringido que el estado inicial de la cola de mensajes sea vacío. Notar también que este tubo actúa de forma activa enviando los mensajes. Así los autómatas que tengan la acción asociada *input recibir(m: Info)* no podrán evitarla si es disparada.

Una opción ante esta situación es la formulación del tubo pasivo que muestra la Fig. 6.

TuboPasivo sólo envía mensajes en la medida que se le solicitan, vía acciones getInfo. Notar, en cualquier caso, que dada la asincronía del modelo IOA, la acción *recibir(m)*, respuesta a una solicitud *getInfo*, no es automática, y puede demorar varios pasos. Es por esto que la variable de estado solicitudes es modelada como de tipo *Int*, más que *Bool*.

```
automaton TuboPasivo (Info : type)
    signature
        input enviar (m : Info),
                getInfo
        output recibir(m : Info)
    states
        mensajes: Seq[Info] := {},
        solicitudes: Int := 0
    transitions
        input enviar(m)
                eff mensajes := mensajes ⊦ m
        input getInfo
                eff solicitudes := solicitudes + 1
        output recibir (m)
                pre solicitudes > 0 ∧
                    mensajes ~= {} ∧
                    m = head (mensajes)
                eff solicitudes := solicitudes - 1;
                    mensajes := tail (mensajes)
```

Fig. 6 Especificación de un tubo pasivo.

Invocación Remota. Aquí queremos crear una componente que pueda bloquearse al ejecutar una acción de salida (la invocación), en espera de una acción de entrada dada, que desbloquea la componente. Durante este tiempo, la componente bloqueada no debiera ejecutar ninguna otra acción.

El modelo IOA no contempla esta capacidad como algo externo a un autómata, es decir, no hay modo de prohibir desde el exterior a un autómata que éste ejecute acciones, si sus variables de estado lo permiten. La opción aquí es modificar internamente el autómata, de modo que él mismo bloquee toda acción hasta recibir la señal de desbloqueo.

Consideremos el autómata de la Fig. 7 que representa una componente genérica.

```
automaton Componente
    signature
        input accion1 (m: Info),   \% desbloquea la comp
              accion2
        output accion3 (m: Info)   \% bloquea la comp
    states
        mi_var: Int
    transitions
        input accion1 (m)
            eff hacer_algo1
        input accion2
            eff hacer_algo2
        output accion3 (m)
            pre precond
            eff hacer_algo3
```

Fig. 7 Especificación de una componente genérica.

Una forma de hacer que este autómata se bloquee luego de ejecutar una acción *accion3(m)*, es agregar a sus variables de estado un *Bool* llamado *bloq*, de modo que cuando esta variable esté en {\em true}, ninguna acción pueda ser lanzada, tal como se muestra en la Fig. 8.

```
automaton ComponenteRPC
signature
    input accion1 (m: Info),   \% desbloquea la comp
          accion2
        output accion3 (m: Info)   \% bloquea la comp
    states
        mi_var: Int,
        bloq: Bool := false
    transitions
        input accion1 (m)
            eff hacer_algo1;
                bloq := false    \% ejecuta y desbloquea
        input accion2
            eff if (~bloq)
               then hacer_algo2 \% ver comentario **
               fi
        output accion3 (m)
            pre (~bloq) ∧ precond
            eff hacer_algo3;
                bloq := true      \% ejecuta y bloquea
```

Fig. 8 Especificación de un autómata con invocación remota.

Como se ve, la conversión de un autómata genérico en un cliente obedece a reglas específicas: agregar la variable de estado *bloq*, agregar *(~bloq)* a las precondiciones de las acciones de salida, y agregar las instrucciones *bloq := false* y *bloq := true* en los efectos de las acciones correspondientes.

El único caso donde la conversión no puede hacerse mecánicamente, sino que requiere una decisión de diseño, es en el caso indicado con ** en la Fig. 8. En este caso, la *ComponenteRPC* recibirá las acciones *input accion2* mientras esté bloqueada, pero no reaccionará ante ellas gracias a la instrucción *if (~bloq)*, ésta fue introducida bajo el supuesto de que efectivamente se desea que el autómata descarte toda acción de input que no sea *accion1(m)* durante su período de bloqueo, lo cual no necesariamente es así: también podría ser deseable que las acciones de entrada que *ComponenteRPC* reciba durante su período de bloqueo queden encoladas a la espera de que la componente retome su actividad.

La especificación de los distintos patrones de arquitectura populares incluidos anteriormente permite ilustrar las distintas características que IOA muestra como lenguaje de descripción de arquitecturas.

RESULTADOS

Componentes en IOA

Las características deseables que debiera tener un ADL para especificar componentes son: interfaz, tipos, semántica, restricciones, evolución y propiedades no funcionales. Si bien los ejemplos incluidos en los patrones de arquitectura en IOA sólo ilustran parte de las capacidades de IOA para la especificación de componentes, la Tabla 1 describe cómo IOA aborda cada una de estas características. Podemos ver que todas ellas son cubiertas si bien algunas de ellas, tales como restricciones y propiedades no funcionales, de manera acotada.

Conectores en IOA

Entender una componente como un autómata parece bastante natural; entender un conector como un autómata es algo más indirecto. Es por eso que en la de patrones de arquitectura en IOA hemos incluido fundamentalmente distintas formas de especificar conectores usando IOA. Es un argumento recurrente en la literatura [1] que el permitir especificar conectores como elementos de modelamiento "de primera clase" es una característica importante y no demasiado difundida entre los ADLs. La Tabla 3 describe cada una de las facetas relevantes en la definición de conectores usando un ADL y cómo IOA las aborda.

Configuraciones en IOA

IOA tiene elementos para especificar todas y cada una de las características deseables de la definición de una configuración usando un ADL. Esto da como resultado que tenga todos los elementos que hacen a la calidad de la descripción de la configuración, la calidad de la descripción del sistema, y sólo en una menor medida a las propiedades de la descripción del sistema. Esto último se debe a que las capacidades de IOA para las especificaciones de propiedades no funcionales no son tan poderosas. La Tabla 4 describe cómo IOA aborda cada una de las características deseables de una definición de configuración o topología.

Tabla 2 Resumen de características de componentes en IOA.

Característica	Implementación en IOA
Interfaces	La signature de los autómatas permite especificar la interfaz de una componente, aunque no los tipos de la interfaz.
Tipos	Las componentes pueden ser paramétricas en los tipos de datos que manejan y también pueden ser definidas con identificadores genéricos que luego se instancian para cada arquitectura concreta posiblemente con distintos valores.
Semántica	La semántica de las componentes está dada por la especificación en un alto nivel de abstracción del comportamiento que tendrá ante distintos eventos, ya sean internos o provenientes del entorno.
Restricciones	Se puede especificar invariantes sobre los valores iniciales de las variables de estado de la componente así como también los rangos entre los cuales puede elegirse los valores de las variables de estado en forma no determinística.
Evolución	Es posible establecer formalmente la relación binaria de simulación hacia adelante (forward simulation) que establece que una componente es una posible implementación de la otra. Esto permite que puedan tenerse más de una implementación de una misma componente especificada en forma abstracta.
Propiedades no funcionales	IOA permite especificar algunas propiedades no funcionales de los autómatas como es el caso de los *tasks:* se puede establecer que ciertas acciones sobre las cuales el autómata tiene control (internas o de output) para evitar situaciones de hambruna (starvation) cuando existen varias acciones habilitadas.

Tabla 3 Resumen de características de conectores en IOA.

Característica	Implementación en IOA
Interfaces	Como los conectores se especifican esencialmente como autómatas, su interfaz se especifica también en su signature.
Tipos	En la sección sobre patrones de arquitectura mostramos una serie de autómatas representando esencialmente distintos tipos de conectores que pueden ser instanciados mediante la asignación de un tipo particular a sus tipos paramétricos y un valor particular a sus identificadores.
Semántica	La semántica de los conectores está dada por el protocolo que siguen para permitir la comunicación entre las componentes. Esta semántica está dada por las acciones que realiza cada vez que recibe un mensaje de input y cuando envía mensajes de output; todo esto se especifica mediante sus transiciones.
Restricciones	Es posible establecer restricciones en los conectores tales como el tamaño máximo de un buffer o de un repositorio de memoria compartida.
Evolución	Al igual que los componentes, también los conectores, al estar especificados como autómatas, permiten la definición de distintas implementaciones de un conector especificado con un alto nivel de abstracción.
Propiedades no funcionales	Las propiedades no funcionales son limitadas y pueden representarse con *tasks* para sincronización de mensajes, lo cual puede ser muy útil para especificar protocolos.

Tabla 4. Resumen de características de configuraciones en IOA.

Características	Implementación en IOA
Comprensibilidad	Si bien los IOA han sido utilizados para la especificación de algoritmos distribuidos altamente complejos, los componentes, conectores y la forma en que se componen para formar configuraciones más sofisticadas siguen una forma sencilla. Al menos podemos decir que los IOA no son más complejos de comprender que ningún otro ADL aún manejando todos los conceptos involucrados.
Composición jerárquica	Dos o más autómatas pueden componerse para formar un autómata más complejo. Luego este autómata puede, a su vez, combinarse con otros autómatas simples o compuestos para formar otros aún más complejos. Esta composición jerárquica es una operación sencilla y natural en los IOA. También es posible lograr un grado mayor de encapsulamiento de autómatas compuestos ocultando algunas operaciones de output, de modo que sólo interactúe con otros autómatas a través de una interfaz más restringida.
Refinamiento y seguimiento	IOA permite establecer que un autómata posiblemente compuesto sea la implementación (refinamiento) de otro especificado en forma más abstracta. Así, este formalismo permite hacer una especificación con un alto nivel de abstracción y refinarla sucesivamente pudiendo hacer un seguimiento del impacto de cada paso a través de las herramientas existentes [8].
Heterogeneidad	IOA tiene incorporada la posibilidad de especificar tipos de datos abstractos usando Larch [10] [11] permitiendo así encapsular la definición de tipos posiblemente complejos y que nada aportan a la definición de la arquitectura del software. Además, IOA permite especificar los efectos de una transición tanto con un predicado lógico como con una acción imperativa como la asignación. Las herramientas de simulación de IOA [8] permiten también poner invocaciones a programas Java como parte de los efectos de una transición.
Escalabilidad	La escalabilidad en IOA tiene estrecha relación con la composición paralela y su aplicación en forma jerárquica. Para que un sistema crezca basta con componerlo con nuevos autómatas que interactúen con el ya definido a través de su interfaz.
Evolucionabilidad	La posibilidad de evolucionar tiene relación con la capacidad para expresar el refinamiento de autómatas. Una familia de sistemas de software puede corresponder a distintas implementaciones de un mismo autómata posiblemente compuesto. Puede también existir sucesivos refinamientos de un mismo autómata.
Dinamismo	Existe un formalismo derivado de IOA llamado Dynamic IOA [3] que permite especificar arquitecturas que cambian durante su ejecución principalmente creando y destruyendo instancias de determinados tipos predefinidos. Este formalismo, si bien es más compacto para especificar dinamismo, está demostrado que es equivalente al poder expresivo de los IOA simples [3].
Restricciones	Las herramientas de análisis y simulación de IOA permiten especificar invariantes que deberá cumplir el autómata compuesto que representa al sistema modelado. De esta forma las restricciones aplican a todo el sistema y no solamente a sus componentes y conectores en forma independiente.
Propiedades no funcionales	Pueden especificarse propiedades no funcionales mediante invariantes de los autómatas compuestos, y también restricciones de sincronización de las distintas acciones de los autómatas de la composición usando *tasks*.

CONCLUSIONES

Pese a que IOA no es un lenguaje originalmente creado para la especificación de arquitecturas de software, hemos mostrado que, de acuerdo con las características que debiera tener para poder ser considerado un ADL, éste cumple satisfactoriamente con cada una de ellas. Muy pocos de los ADLs más populares tienen esta propiedad. IOA cuenta además con una serie de herramientas de apoyo que lo hacen aún más usable, lo cual unido a su poder expresivo y la posibilidad de generar código ejecutable, lo hace un lenguaje muy promisorio para el uso práctico de especificación de arquitecturas.

Hemos usado IOA en combinación con diagramas informales de cajas y líneas para especificar arquitecturas

[3] y hemos encontrado que resultan, además de completamente formales, una notación usable y no muy compleja, ni para desarrollar especificaciones ni para comprenderlas.

Actualmente estamos estudiando la posibilidad de aplicar IOA también para la especificación de arquitecturas de líneas de productos de software. Dadas sus características de apoyo a la especificación de tipos, evolución, refinamiento y seguimiento y escalabilidad, IOA es un buen candidato para especificar las variaciones típicas de este tipo de arquitecturas.

REFERENCIAS

[1] G.D. Abowd, R. Allen and D. Garlan. "Formalizing style to understand descriptions of software architecture". ACM Transactions on Software Engineering and Methodology, 4(4):319-364, 1995.

[2] R. Allen and D. Garlan. "A Formal Basis for Architectural Connection". ACM Transactions on Software Engineering and Methodoly, 6(3):213-249, July 1997.

[3] P. Attie and N. Lynch. "Dynamic Input/Output Automata: a Formal Model for Dynamic Systems". In CONCUR'01, the International Conference on Concurrency Theory, Aalborg, Denmark, August 2001.

[4] M. C. Bastarrica, S.F. Ochoa and P.O. Rossel. "Integrated Notation for Software Architecture Specification". In Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC), Arica, Chile, November 2004. IEEE Press.

[5] F. Buschmann, R. Meunier, H. Rohnert and P. Sommerlad. "Pattern Oriented Software Architecture: A System of Patterns". John Wiley & Son Ltd., August 1996.

[6] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford. "Documenting Software Architectures. Views and Beyond". SEI Series in Software Engineering. Addison Wesley, 2002.

[7] D. Garlan, Shang-Wen Cheng and A.J. Kompanek. "Reconciling the Needs of Architecture Description with Object-Modeling Notations". Science of Computer Programming, 44(1):23-49, July 2002.

[8] S. Garland and N. Lynch. "Using I/O Automata for Developing Distributed Systems. in Foundations of Component-Based Systems", G. T. Leavens and M. Sitaraman, Eds., chapter 13, pp. 285-312. Cambridge University Press, USA, 2000.

[9] S.J. Garland and N.A. Lynch. "The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems". Technical Report MIT/LCS/TR-762, MIT Laboratory for Computer Science, Cambridge, MA, August 1998.

[10] J.V. Guttag, J.J. Horning, and J.M. Wing. "The Larch Family of Specification Languages". IEEE Software, 2(5), 1985.

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

[12] I. Jacobson, G. Booch, and J. Rumbaugh. "The Unified Software Development Process". Object-Technology Series. Addison-Wesley Profesional, February 1999.

[13] N. Lynch. "Distributed Algorithms". Morgan Kaufmann Publishers, 1996.

[14] N. Lynch and M. Tuttle. "An Introduction to Input/Output Automata". CWI Quart, 2(3):219-246, 1989.

[15] N. Medvidovic and R. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages". IEEE Transactions on Software Engineering, 26(1):70-93, January 2000.

[16] Mary Shaw and David Garlan. "Software Architecture. Perspectives on an Emerging Discipline". Prentice Hall, 1996.