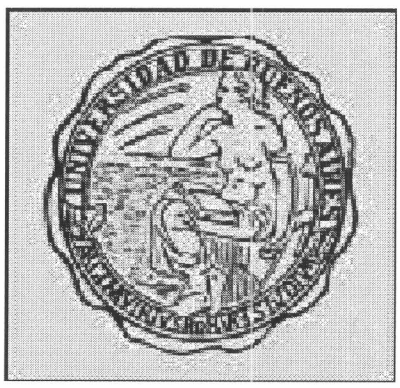
Tesis de Licenciatura

Definición e Implementación de un Lenguaje de Descripción de Arquitecturas de Software



Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Tesistas
Galiana, Román
Medrano, Ramiro
Mohnen, Laura

Director
Lic. Kicillof, Nicolás

2006

Indice

1 INTRODUCCIÓN 4

1.1 OBJETIVOS 4

1.2 CONTEXTO 4

2 FUNDAMENTOS DE LA INVESTIGACIÓN 5

2.1 ARQUITECTURAS DE SOFTWARE 5

2.1.1 Introducción 5

2.1.2 Breve descripción del proceso técnico en arquitecturas 6

2.2 LENGUAJES DE DESCRIPCIÓN DE ARQUITECTURAS (ADLS) 6

2.2.1 Introducción 6

2.2.2 Clasificación de ADLs y marco de comparación 7

2.2.3 Modelando componentes 8

2.2.4 Modelando conectores 8

2.2.5 Modelando configuraciones 9

2.2.6 Soporte de herramientas para la descripción arquitectural 9

2.3 REPRESENTACIÓN SEMÁNTICA EN EL NIVEL DE COMPORTAMIENTO 9

2.3.1 Utilización de Redes de Petri en JACAL 2.0 10

2.4 MIDDLEWARE 13

2.5 XML 14

2.6 PARADIGMA DE COMUNICACIÓN PUBLISH/SUBSCRIBE 14

3 JACAL 15

3.1 INTRODUCCIÓN AL LENGUAJE 15

3.1.1 Ventajas de utilizar JACAL 15

3.1.2 Nivel de interfaz 16

3.1.3 Nivel de Comportamiento 19

3.2 ADAPTACIÓN DEL LENGUAJE: CONSTRUCCIÓN DE JACAL 2.0 19

3.3 HERRAMIENTA JACAL 2.0 24

3.4 COMPARACIÓN CON OTROS ADLS 26

3.4.1 Acme 26

3.4.2 MetaH AADL 27

3.4.3 Rapide 27

3.4.4 UniCon 28

3.4.5 Wright 28

4 CASOS DE ESTUDIO 29

5 CONCLUSIONES 35

6 CONTRIBUCIONES 36

7 POSIBLES EXTENSIONES 36

8 REFERENCIAS 37

Definición e Implementación de un Lenguaje de Descripción de Arquitecturas de Software

1 INTRODUCCIÓN

1.1 Objetivos

- Definir un Lenguaje de Descripción de Arquitecturas de Software (ADL) de propósito general, adaptado conceptualmente a las tendencias actuales en la materia y que permita simular la ejecución de modelos y transformar recíprocamente la especificación textual de éstos y su representación gráfica
- Implementar, para tal efecto, una herramienta de edición gráfica (aprovechando tecnologías existentes), la importación/exportación entre los distintos formatos y un simulador de ejecuciones de sistemas basados en las arquitecturas escritas en el lenguaje

1.2 Contexto

Nuestro principal objetivo, entonces, es especificar un ADL, y con él, un entorno de diseño que dé soporte a su utilización, aprovechando la base conceptual actual ([14], [15], [25]) y proveyendo como resultados tanto un lenguaje sólido y portable como una herramienta innovadora.

Para esto, nuestro esfuerzo se concentró en usar como base del trabajo un lenguaje de este tipo ya existente llamado JACAL (objeto de otras 2 Tesis de Licenciatura a la fecha: [2], [3]) y mejorar su semántica y sus características generales de modo de poder alcanzar nuestros Objetivos propuestos. Con ellos en mente es que creamos ***JACAL 2.0***.

JACAL es un ADL que tiene la ventaja comparativa de permitir expresar la suficiente información sobre el comportamiento interno de las partes constituyentes de una arquitectura como para que se pueda construir un modelo dinámico de un sistema, simular su ejecución y brindar información útil para potenciales herramientas que puedan usufructuarlo (ej: model-checking automático).

Una arquitectura en JACAL se construye describiendo los llamados *nivel de interfaz* (que muestra qué componentes y conectores tiene el sistema, cuáles son sus interfaces y cómo ocurren las comunicaciones entre ellas) y *nivel de comportamiento* (que muestra los cambios del estado interno de un componente cuando recibe algún input de otro, pudiendo, a su vez, mandar señales a otros componentes) esperados en un sistema.

Para entender la decisión de trabajar sobre JACAL (e intentar mejorarlo), creemos necesario considerar primero un breve resumen del marco teórico que sustenta la definición de un ADL y de los conceptos involucrados (Arquitectura de Software, Middleware, componentes, conectores, etc), y luego detallar nuestro enfoque particular.

A continuación, brindamos una aproximación introductoria a cada uno de estos temas, para luego ingresar en la definición en sí del lenguaje propuesto.

2 FUNDAMENTOS DE LA INVESTIGACIÓN

2.1 Arquitecturas de Software

2.1.1 Introducción

No existe una definición estándar, universalmente aceptada, acerca del alcance que debe tener una arquitectura de software.

Una arquitectura de software ha sido definida por distintos autores ([18], [27]) como:

- La selección de los elementos estructurales y sus interfaces con las que se compone el sistema.
- El comportamiento de los elementos estructurales especificado como colaboración entre esos elementos.
- La composición de esos elementos estructurales y su comportamiento en subsistemas progresivamente más grandes.
- La/s estructura/s del sistema, que comprende/n elementos de software, las propiedades externamente visibles de esos elementos, y las relaciones entre ellos.

Nosotros trabajamos bajo una definición de arquitectura de software que nos pareció abarcadora de todas las anteriores, y que es similar a la utilizada por [4] y [5], a saber:

- Un nivel de diseño que involucra la descripción de los elementos estructurales sobre los cuales se construyen los sistemas (*componentes*), las interacciones entre esos elementos (*conectores*), los patrones que guían su composición y las restricciones sobre esos patrones.

Analizando un poco más exhaustivamente la definición anterior, vemos que:

- Todo sistema de software puede ser especificado en su nivel de arquitectura, dado que todo sistema puede ser mostrado como compuesto de *componentes* y relaciones entre ellos (a través de los conectores).
- La arquitectura define *componentes* en función de su interacción (del uso de conectores). Esto significa que la arquitectura omite información sobre los *componentes* que no corresponda a esta interacción.
- Respecto del comportamiento de los *componentes* que sí estarán incluidos en la arquitectura, se plasmarán sólo aquellos aspectos que regulen la manera en que esos *componentes* interactúan con otros, por medio de los conectores que se le definan.

- La arquitectura de software debe abstraer alguna información del sistema con el fin de proveer suficiente información para el análisis, toma de decisiones y minimización de riesgos.

2.1.2 Breve descripción del proceso técnico en arquitecturas

Siguiendo los pasos especificados en [18], coincidimos en que el proceso técnico para la creación de una buena arquitectura incluye pasos y heurísticas determinadas. El principal entregable de este proceso es la *especificación arquitectural*, que describe la estructura del sistema a través de varias vistas. De todas formas, aunque la estructuración del sistema es lo principal del proceso, es sólo una de las varias actividades críticas en la creación de una buena arquitectura.

2.1.2.1 Estructuración del sistema

La arquitectura se crea y documenta en la fase de estructuración del sistema, que a su vez se descompone en subfases.

Primero se formula la *visión arquitectural* para que actúe como guía de decisiones durante el resto del proceso en la cual se explicita la meta-arquitectura. Esto incluye el *estilo arquitectural*: conceptos, mecanismos y principios que guiarán la arquitectura durante los próximos pasos.

Se descompone el sistema en componentes, responsabilidades de cada componente e interconexiones entre componentes.

La idea de la arquitectura conceptual es enfocarse hacia una correcta descomposición del sistema sin entrar en detalles acerca de la especificación de las interfaces ni del tipo de información. La arquitectura conceptual constituye el punto de partida para la arquitectura lógica, y puede modificarse y/o refinarse durante su creación. El modelado del comportamiento dinámico del sistema es una manera útil de depurar las responsabilidades y las interfaces de los componentes, cuya especificación permite ir concretando la arquitectura.

2.1.2.2 Validación de la arquitectura

Una fase de validación provee indicadores claros y, por lo tanto, una oportunidad para resolver problemas en la arquitectura. Involucra un modelado de escenarios que representen requerimientos funcionales y el desarrollo de prototipos de prueba.

El proceso de diseño de la arquitectura de un sistema se conduce iterativamente, con múltiples ciclos a través de los requerimientos, estructuración y validación.

2.2 Lenguajes de Descripción de Arquitecturas (ADLs)

2.2.1 Introducción

Para soportar el desarrollo basado en una arquitectura de software que cumpliera con las características descriptas en el capítulo anterior, se hizo necesaria una notación de

modelado de estas arquitecturas: ése es el origen de los ADLs. Análogamente a lo que ocurre con las arquitecturas de software, hay poco consenso sobre qué es un ADL, qué aspectos de una arquitectura debe ser modelada en un ADL y cuál de varios posibles ADLs es mejor para un problema particular.

Un ADL para aplicaciones de software se enfoca en la estructura de alto nivel de la aplicación en su conjunto antes que en los detalles de implementación de algún módulo específico.

Para nosotros, coincidiendo en parte con el panorama descripto en [4] y [5], un ADL debe tener una sintaxis simple, entendible y gráfica (pero formal), y una semántica que permita incluir herramientas como model checkers, herramientas de soporte en tiempo de ejecución, etc.

Un ADL debe modelar explícitamente *componentes*, *conectores* y sus *configuraciones*. Más aún, para ser realmente usable y útil, debe proveer herramientas de soporte para el desarrollo y la evolución basado en arquitectura.

2.2.2 *Clasificación de ADLs y marco de comparación*

Un ADL es, por lo tanto, un lenguaje que provee características para el modelado de la arquitectura conceptual de un sistema de software, distinguiéndola de la implementación del sistema. Los ADLs proveen tanto una sintaxis concreta como un marco conceptual para caracterizar arquitecturas. El marco conceptual refleja características del dominio para el cual se usa el ADL y/o el estilo arquitectural. El marco de trabajo abarca la teoría semántica subyacente (CSP, redes de Petri, máquinas de estados finitos, etc.).

En particular, para este trabajo nos interesaba enfocamos en sostener el concepto de lenguaje *de propósito general*.

Una descripción arquitectural se construye en base a componentes, conectores y configuraciones arquitecturales. Un ADL debe proveer los significados para su especificación explícita; esto nos permite distinguir cuándo una notación es un ADL o no. Las interfaces de los componentes constituyentes deben también ser modeladas. Sin esta información, la descripción arquitectural es sólo un grupo de identificadores, sin una semántica explícita subyacente. Más allá de la flexibilidad que implica para un lenguaje dado el ser soportado por más de una herramienta, un conjunto de herramientas que lo acompañen convierten al ADL en un lenguaje más usable y útil. Como modelo de un sistema en un alto nivel de abstracción, se usa un ADL para proveer una descripción parcial del sistema. Los tipos de información en los que se enfoca el ADL pueden ser las características de un dominio de aplicación, un estilo de composición de sistemas (un estilo arquitectural), o un conjunto específico de propiedades (distribución, concurrencia, seguridad, etc.).

2.2.3 Modelando componentes

Un componente en una arquitectura es una unidad de cómputo o un almacenamiento de datos. Los componentes pueden ser tan pequeños como un simple procedimiento o tan grandes como una aplicación entera. Cada componente puede requerir su propio espacio de datos y ejecución o puede compartirlos con otros componentes. Una característica requerida en un ADL es la definición explícita de la interfaz del componente (o puntos de interacción entre ese componente y el exterior). Las características adicionales de comparación permiten modelar, por ejemplo, tipos de componentes, semánticas (o modelo de alto nivel del comportamiento del componente) y restricciones.

2.2.4 Modelando conectores

Los conectores son los bloques de construcción arquitecturales usados para modelar las interacciones entre componentes y las reglas que gobiernan esas interacciones. Las características principales inherentes a los conectores también son su interfaz (o puntos de interacción entre ese conector y los componentes), tipos (abstracciones que encapsulan la comunicación entre componentes, la coordinación y las decisiones de mediación), semánticas (o modelo de alto nivel del comportamiento del conector) y restricciones.

Los ADLs deben soportar el modelado de semántica en los conectores con el fin de habilitar el análisis de la interacción de componentes, refinamiento consistente de las arquitecturas a través de niveles de abstracción, y poder forzar las restricciones de interconexión y comunicación.

Los sistemas de software están frecuentemente compuestos de componentes prefabricados de naturaleza heterogénea que proveen funcionalidad de cierta complejidad y se ven involucrados en una amplia gama de diferentes interacciones.

La interacción de los componentes está representada en el Nivel de Arquitectura por los conectores de software.

Un paquete de middleware provee un conjunto predefinido de capacidades de interacción de software difícil de extender. Esta tecnología asume un entorno homogéneo en el cual todos los componentes adhieren a un determinado diseño, implementación y restricciones en tiempo de ejecución.

La visión basada en arquitecturas de software separa la computación (componentes) de la interacción (conectores) en un sistema. En principio, las arquitecturas no asumen homogeneidad de componentes ni restringen los conectores permitidos ni sus mecanismos de implementación.

Contar con una taxonomía de conectores de software facilita la tarea de escoger un conector adecuado para cada necesidad. (En particular, la taxonomía utilizada en el presente trabajo está basada en [2] y [6]).

Algunos estilos de arquitecturas motivados por conectores de software son: Pipes & Filters, consumo de datos en tiempo real, arquitectura guiada por eventos, basada en

mensajes y basada en flujo de datos. Cada estilo permite una considerable flexibilidad en la elección de la implementación de sus conectores, lo cual requiere la identificación de los posibles parámetros de variación.

Los conectores median interacciones entre componentes; esto es, establecen las reglas que gobiernan la interacción de componentes y especifican cualquier mecanismo auxiliar requerido.

2.2.5 Modelando configuraciones

Las configuraciones arquitecturales o topologías son grafos conectados de componentes y conectores que describen la estructura arquitectural. Esta información es necesaria para determinar cuándo los componentes están conectados de manera apropiada, si sus interfaces se complementan, si los conectores habilitan apropiadamente la comunicación, y si su semántica combinada resulta en un comportamiento deseado.

2.2.6 Soporte de herramientas para la descripción arquitectural

La motivación detrás del desarrollo de ADLs formales es que su formalidad los hace apropiados para el razonamiento y la manipulación por herramientas de software. Las herramientas de soporte no son, estrictamente hablando, parte de un ADL. De todas formas, la utilidad de un ADL está muy relacionada con el soporte para el diseño, análisis, evolución, generación de un sistema ejecutable, etc.

2.3 Representación semántica en el Nivel de Comportamiento

Para especificar el Nivel de Comportamiento en JACAL 2.0, necesitábamos una herramienta cómoda, fácil de comprender, de uso extendido y eficacia comprobada, de semántica simple y sencilla, y con la suficiente potencia expresiva de representación.

En la versión inicial de JACAL se usaron las Redes de Petri originales ([26]), que cumplían con todos los requisitos mencionados anteriormente, con la excepción parcial del último: claramente algunas arquitecturas con ciertas particularidades no pueden ser representadas con el modelo común de Redes de Petri, como lo que ocurría cuando un mismo componente recibía por el mismo port más de un conector, en cuyo caso se tenía un problema en la identificación (origen) de cada flujo.

Con el objeto de subsanar este tipo de inconvenientes en la representación de arquitecturas es que para JACAL 2.0 utilizamos la notación proporcionada por las Redes de Petri Coloreadas ([21], [22], [23]), que nos permiten diseñar el comportamiento que esperamos de arquitecturas más complejas, y construirlas.

Al manejar representaciones con Redes de Petri Coloreadas, en JACAL 2.0 también estamos incluyendo el manejo de tipos de datos que conlleva; con esta facilidad, los colores pueden asociarse a estos tipos de datos, que pueden contener información implícita tal como

el port de input o el conector al que responder en un *call*, e incluso permitir sentencias condicionales, como así también una representación de pila para poder seguir la secuencialidad de los tokens en aquellos casos en los que varios conectores se conectan al mismo port de input y respetar esa secuencia para la respuesta ordenada. En particular, el tipo de datos que utilizamos para esta semántica (al que llamamos *Flow*) consta de una pila de ports, una pila de conectores y un valor parámetro -secuencia de caracteres-. (Para un mayor detalle, ver ***Anexo III – Semántica de Ports y Conectores***, que acompaña y complementa a este documento).

La facilidad originada en la inclusión de tipos de datos es una gran ventaja con respecto al JACAL original, que en los casos de return implícito la semántica permitía especificar, pero el comportamiento real no se correspondía con dicha especificación.

2.3.1 Utilización de Redes de Petri en JACAL 2.0

Aprovechando las facilidades de especificación formal de la semántica y su representación que otorgan las Redes de Petri, así como su amplia difusión y conocimiento general, plasmamos el comportamiento de los distintos elementos constitutivos de JACAL 2.0 utilizando esta poderosa herramienta.

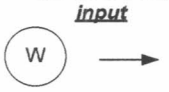
Utilizando Drawings de MS Visio, fuimos detallando los Tipos de Ports que definimos, las relaciones permitidas entre los Tipos de Ports y los Tipos de Componentes, los distintos *places* ad hoc que definimos para cada Tipo de Port, la representación (en notación Petri) asociada al comportamiento de cada Tipo de Port en cada Tipo de Componente permitido, y finalmente la representación (en notación Petri) del comportamiento de cada Tipo de Conector que definimos para JACAL 2.0.

A continuación sólo se muestran, a modo de ejemplo, las definiciones acordadas para Tipos de Ports, el detalle de la semántica del Port de Input y el detalle de la representación del comportamiento de los conectores **call** y **queue** (la información restante se encuentra en el ***Anexo III – Semántica de Ports y Conectores***, que acompaña y complementa al presente documento):

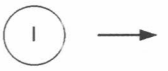
- El tipo de Port asociable a un componente depende del Tipo del mismo:
  - Componentes reentrantes → Ports de Tipo Input o de Tipo Output
  - Componentes repositorios → Ports de Tipo Read o de Tipo Input

- Para determinar sus comportamientos ante cada posible transición, se definen determinados places para cada uno de ellos, a saber:

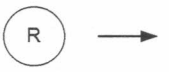
Ports de tipo input



Almacenamiento persistente de tokens en espera

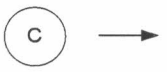


Destino de tokens entrantes

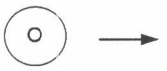


Retorno de tokens

Ports de tipo output

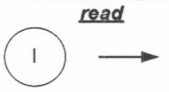


Recepción de continuación

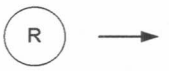


Destino de tokens salientes

Ports de tipo read



Destino de tokens entrantes



Retorno de tokens

Ports de input

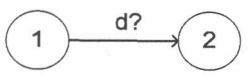
Especificación del comportamiento asociado a las transiciones

Se asume que todos los lugares de las redes de Petri definidas tienen asociado el tipo de dato Flow:
(List<Port> x List<Connector> x Data)

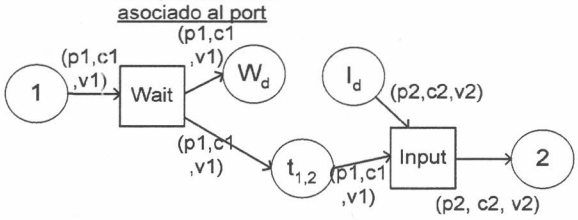
Nivel de comportamiento

Comportamiento asociado al port

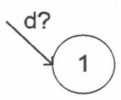
I)



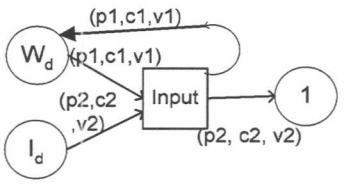
(transición asociada al port d)



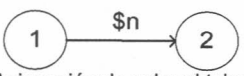
II)



Componente reentrante



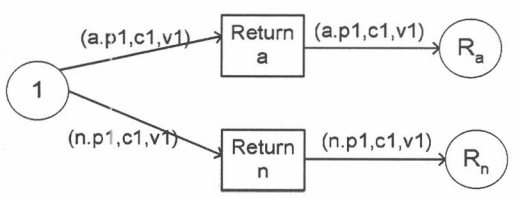
III)



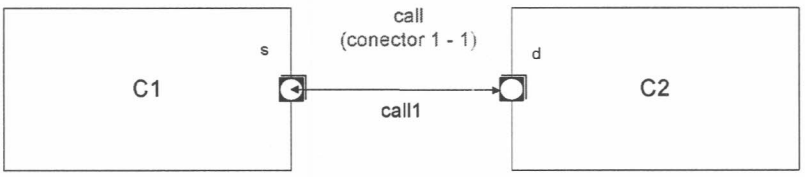
(Asignación de valor al token. El estado 2 puede o no ser terminal)

IV)

(estado terminal)

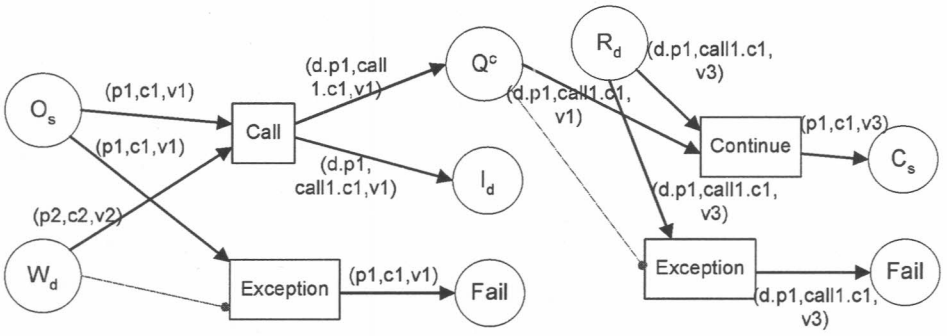


Nivel de interfase

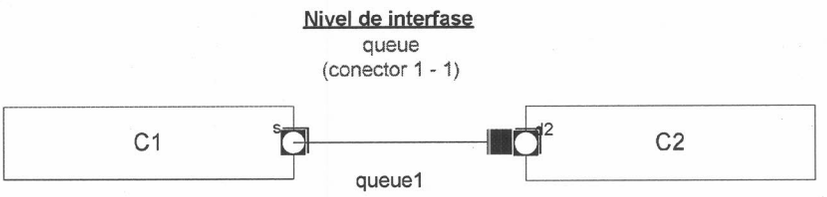


Dinámica pretendida:
1) C1 envía mensaje por *call* a C2.
2) C2 empieza a ejecutar.
3) C2 devuelve resultado a C1, que continúa ejecutando de acuerdo con el valor devuelto.

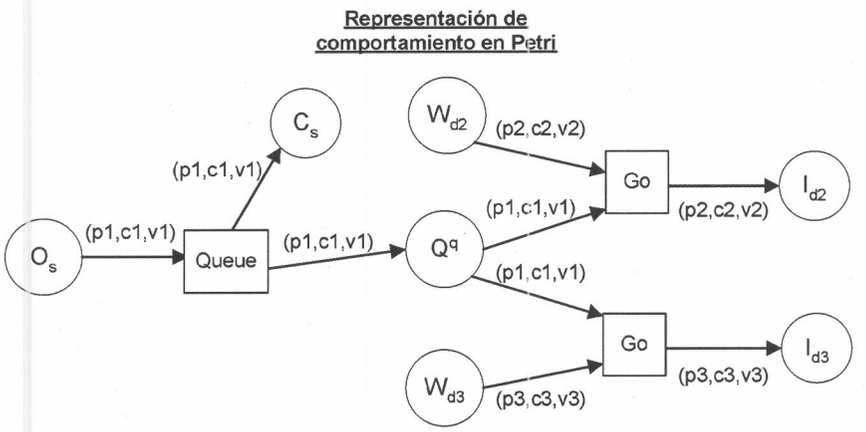
Representación de comportamiento en Petri



(Ejemplo de representación de conector **call**)



Dinámica pretendida:
1) C1 envía mensaje por *queue* a C2.
2) Simultáneamente:
*) El mensaje es dejado en la cola central;
*) C1 continúa ejecutando.
[3) En el momento en que algún receptor -aquí se ejemplifica con C2- consulte la cola:
a) si el mensaje estaba y él es el único que lo estaba esperando, lo consumirá; o
b) si el mensaje estaba y hay más componentes esperándolo, se decidirá no determinísticamente quién lo consumirá (los no elegidos seguirán bloqueados); o bien
c) si no había ningún mensaje, se queda bloqueado hasta que lo haya.
4) Luego de consumir el mensaje, C2 continúa ejecutando]



(Ejemplo de representación de conector **queue**)

2.4 *Middleware*

El middleware es una clase de tecnología de software diseñada para ayudar a manejar la complejidad y heterogeneidad en sistemas distribuidos. Es una capa de software sobre el sistema operativo pero bajo el programa de aplicación, que provee una abstracción de programación común sobre un sistema distribuido.

Para este trabajo, y siguiendo la nomenclatura de [7] y [9], la categoría de middleware que utilizaríamos en nuestro trabajo sería **Middleware Orientado a Mensajes (MOM)**, que provee una conexión entre aplicaciones, basada en el intercambio de mensajes. Este tipo de middleware se caracteriza por ofrecer separación temporal, ser asincrónico y espacial, y en él los procesos no necesitan conocer cada una de las identidades.

Por ejemplo, para nuestro trabajo estamos usando el modelo MOM básico de **Publish-subscribe**, que permite transmitir mensajes a múltiples receptores y usualmente usar

multicasting como transporte subyacente. Los mensajes son enviados a una cola particular a la cual los clientes pueden suscribirse.

2.5 XML

XML (eXtensible Markup Language) es un meta-lenguaje que permite definir lenguajes de marcado, adecuados a usos determinados. Los objetivos de su diseño fueron: alta usabilidad sobre Internet, soporte a una amplia variedad de aplicaciones, compatibilidad con SGML, facilidad de escritura de programas que procesen documentos XML, minimización de características opcionales, generación de documentos XML legibles y razonablemente claros, diseño formal y conciso ([10], [11], [19]).

El estándar XML incluye un conjunto de estándares asociados, entre los cuales están XML Linking Language (XLink) y Document Object Model (DOM). Las facilidades de XLink permiten establecer relaciones unidireccionales o bien bidireccionales entre documentos XML. DOM es una interfaz independiente del lenguaje, que da a las aplicaciones acceso por programa a un documento XML.

La descripción de la información de una forma estándar en XML permite compartir ampliamente los modelos, y al momento de la especificación de un lenguaje permite expresar todas las estructuras a considerar, especificar propiedades y atributos, organizar las relaciones existentes entre entidades y proveer un marco para futuros cambios en la definición, facilidad de traducción de representaciones y captura de la información necesaria para soportar múltiples vistas de representación (ya que la información puede ser encapsulada en tags, o vinculada desde o hacia un destino usando las facilidades de vinculación de XML). Además, XML es el estándar emergente para repositorios en el mercado, lo cual extiende las facilidades al almacenamiento persistente de modelos, haciéndolos accesibles desde una variada gama de aplicaciones.

2.6 Paradigma de comunicación Publish/Subscribe

Publish/Subscribe (pub/sub) es considerado uno de los estilos de interacción más importantes en el explosivo mercado de integración de aplicaciones corporativas. Los productores publican información sobre un bus de software y los consumidores suscriben a la información que quieren recibir de ese bus. Las aplicaciones pub/sub están organizadas como una colección de componentes autónomos que interactúan publicando eventos y suscribiendo a las clases de eventos en los cuales están interesados ([12], [13], [16]).

El modelo de comunicación y coordinación que resulta de este esquema es inherentemente asincrónico; multi-punto, dado que los eventos son enviados a todos los componentes interesados; anónimo, porque la identidad del emisor se oculta al receptor; implícito, porque el conjunto de receptores de cada evento es elegido implícitamente basado

en las suscripciones y no puede ser modificado por el emisor; y sin estado, dado que los eventos no son almacenados en forma persistente por el sistema luego de recibidos.

Estas características resultan en un fuerte desacoplamiento entre los publicadores de eventos y los suscriptores, lo cual reduce sustancialmente el esfuerzo requerido para reconfigurar la arquitectura de la aplicación en tiempo de ejecución de forma que responda a diferentes tipos de cambios en el entorno externo.

La naturaleza de bajo acoplamiento de la interacción entre los publicadores y los suscriptores no es solo importante para los productos corporativos sino también para aplicaciones de e-commerce, telecomunicaciones y entornos wireless.

Con el surgimiento de las redes de área extendida, la importancia de mecanismos de comunicación eficientes, flexibles y bien estructurados es cada vez mayor. Basar una interacción compleja entre múltiples hosts sobre modelos de comunicación individual punto a punto es una complicación para el desarrollador y conlleva a un modelo de aplicaciones estático y limitado. Más aún, en comunicaciones móviles, puede no ser sencillo para una aplicación descifrar la ubicación exacta de un componente en todo momento. Además, el número de entidades interesadas en cierta información varía a través del ciclo de vida completo de un sistema. Todas estas restricciones demandan un modelo de comunicación flexible, reflejando la naturaleza dinámica de las aplicaciones. El estilo de interacción pub/sub ha probado su eficacia para cubrir ese espacio. El desacoplamiento de las partes tanto en tiempo como en espacio es una clave para la escalabilidad.

3 JACAL

3.1 Introducción al lenguaje

Creado por Nicolás Kicillof en su trabajo de Tesis de Licenciatura en Ciencias de la Computación (FCEN-UBA, [2]), bajo la dirección de Daniel Yankelevich, JACAL define un lenguaje y una notación gráfica que pueden usarse para representar la arquitectura de un sistema con su modelo dinámico. Estas herramientas permiten el desarrollo de una aplicación de software tal que un usuario pueda dibujar una especificación JACAL y ejecutarla, exponiendo el comportamiento de un sistema en términos de la interacción de sus componentes, ayudando a evitar más desarrollo sobre un mal diseño. Integra, por lo tanto, un ADL con un entorno gráfico de diseño en el cual se construye un prototipo en funcionamiento que permite modelar la ejecución concurrente de sus componentes.

3.1.1 Ventajas de utilizar JACAL

- Se supone que mientras se planifica la arquitectura de un sistema, el diseñador tiene en mente un modelo dinámico del sistema, y que algunas de las propiedades que se espera que se cumplan en este modelo pueden probarse antes que el proceso de

desarrollo de software vaya más allá de esta etapa inicial. Por lo tanto, un lenguaje para arquitecturas de software debería permitir al diseñador representar la suficiente información como para hacer esta prueba.

- Se considera aquí que una descripción suficientemente detallada como para proveer la información necesaria para la prueba es parte de la arquitectura del sistema (*cómo* deberían interactuar dos componentes del sistema).
- Análisis de tipos de conectores para que el diseñador pueda encapsular patrones de comunicación conocidos en constructores del lenguaje (en lugar de representarlos como componentes unívocos de un sistema particular).
- Soporte de notación gráfica para representar arquitecturas.
- Formalidad (precisión): el lenguaje debe poseer una semántica formal. La función que mapea una arquitectura a su significado debería ser tan cercana como sea posible a una correspondencia 1:1, y el modelo resultante debería tener una cercana semejanza al lenguaje en sí.
- Los distintos ADLs usualmente representan la comunicación potencial entre componentes, pero se necesita una relación de causalidad para determinar cómo y cuándo una comunicación real disparará otras. Esto sólo puede alcanzarse por una combinación de sintaxis y semántica que mapea arquitecturas a modelos dinámicos.

Niveles de descripción de JACAL ejecutable:

- El nivel alto (nivel de interfaz) muestra qué componentes tiene el sistema, cuáles son sus interfaces y cómo ocurren las comunicaciones entre ellas.
- El nivel bajo (nivel de comportamiento) muestra los cambios del estado interno de un componente cuando recibe algún input de otro, pudiendo, a su vez, mandar otras señales a uno o más componentes.

3.1.2 Nivel de interfaz

- Define componentes y conectores
- Aquí, el comportamiento de la arquitectura *sólo está implícito* en los tipos de conectores y componentes elegidos por el diseñador como bloques de construcción

Los **componentes** se declaran especificando su tipo e interfaz.

- **Tipo**: restringen los contextos en los que puede usarse un componente
- **Interfaz**: conjunto de ports Input/Output y ports de tipo Read, a los que se adosan los conectores. Los ports de tipo Output deben aparecer en a lo sumo una interfaz de componente. Los ports de tipo Input pueden ser usados por más de un componente. Y los ports de tipo Read se asocian a componentes de tipo repositorio.

Un tipo de componente puede tener uno o más subtipos, que son también tipos de componentes.

Un tipo puede tener más de un supertipo, heredándolos (un componente de un tipo es también un componente de todos sus supertipos, etc.)

(Si se crea un tipo nuevo, sólo deben darse explícitamente sus características específicas, ya que el resto se hereda de sus supertipos).

Los **conectores** se declaran especificando su tipo e interfaz:

- **Tipo**: tiene una *aridad* asociada (relación binaria sobre el conjunto de todos los posibles tipos de componente). Al definirla, se restringen los contextos en los que un conector de ese tipo puede ser usado.

Las aridades para tipos de conectores están implícitamente declaradas en la definición de tipos de componentes.

Ej.: "Tipo de componente A puede conectarse a componentes de tipo B por un conector de tipo c" ≡ "(A;B) está en la aridad de c"

Tipos de conectores:

- ***call***: transfiere el control a un componente, y espera que se le devuelva un resultado.
- ***message***: ubica el mensaje en n colas (una para cada componente destino), y continúa su ejecución.
- ***interrupt***: interrumpe todo flujo de control actual en el componente receptor, y espera un resultado.
- ***fork***: agrega un flujo de control a cada receptor; el emisor continúa ejecutando.
- ***kill***: mata todo flujo de control en cada receptor; el emisor continúa ejecutando.
- ***datagram***: cada receptor recibe el mensaje sólo si está esperando una comunicación a través de este conector (si no, el mensaje se pierde). En cualquier caso, el emisor continúa ejecutando.
- ***read***: la ejecución en el emisor continúa de acuerdo con el estado del receptor.
- ***write***: cambia el estado de los receptores; el emisor continúa ejecutando.

Interfaz: Se especifica declarando el nombre junto al par de ports que conecta (el orden representa la dirección de flujo de la información); alternativamente, podría definirse con el caracter '#' en el destino (para señalar que el componente está generando un *evento*) o en el origen (para señalar que el componente maneja un *evento*).

JACAL no permite conectores n:m; una comunicación simétrica de dos vías se representa por dos conectores, uno en cada dirección. Una comunicación a través de un conector es una señal desde un componente a un conjunto de componentes (que podrían, a su vez, mandar una respuesta), formado por todos los componentes que comparten el port Output del conector en su interfaz. La cardinalidad de este conjunto está determinada por el tipo del

conector. Se utilizan conectores 1:n para representar multicast. (Si se usaran n conectores 1:1, habría que definir un orden arbitrario de realización de estas comunicaciones).

Eventos: la gestión de *eventos* es el mecanismo utilizado en el JACAL original para la representación del paradigma de comunicación Publish / Subscribe introducido en 2.6.

Cuando un conector tiene '#' (como destino o como origen), el nombre asignado a su interfaz es el nombre del evento; cuando se genera un evento, los componentes que manejan eventos a través de conectores con igual nombre que el usado en la generación actuarán como destinos para la comunicación. Origen y destino deben usar el mismo tipo de conector para establecer la conexión. En realidad, la semántica de manejo de eventos está estrechamente ligada al tipo de conector usado:

- ***call***, ***interrupt*** → Como sus conexiones son 1:1, si más de un manejador de eventos está presente en el sistema, sólo uno (elegido no determinísticamente) recibirá la comunicación.
- ***message*** → Si el sistema contiene más de un componente con un port conectado a un conector de tipo ***message*** con el mismo nombre que el usado para mandar el msg a '#', el comportamiento será el mismo que en multicasts (o sea que se asumirá una cola individual para cada componente)
- ***fork***, ***datagram***, ***kill*** → El comportamiento es similar al de multicasts (o sea que todo componente que maneje el evento recibirá la comunicación)
- ***read***, ***write*** → No pueden usarse para generar o manejar eventos.

(Permitir el disparo de eventos a través de diferentes tipos de conectores incrementa la flexibilidad y potencia expresiva de JACAL, dando al diseñador la posibilidad de representar muchos tipos de eventos).

Tipos de componentes: se usan en combinación con la aridad de los tipos de conectores para restringir las arquitecturas posibles que pueden construirse (ej: un dispositivo de almacenamiento no debería recibir un flujo de control -a pesar de que su driver podría- porque tal diseño nunca podría implementarse).

- ***no reentrante*** → no acepta ***call***, ***fork*** (mantienen un flujo de control unívoco), ***read*** ni ***write***.
- ***pasivo*** → sólo acepta ***read*** y ***write***; todo cambio de estado debe realizarse por otro componente (no ***pasivo***). Su estado no puede cambiar vía transiciones no visibles desde afuera del sistema (transiciones τ)
- ***reentrante*** → no recibe ***read*** ni ***write***; a pesar de que no es probable, un ***interrupt*** podría emitirse hacia un componente ***reentrante*** (ej: un componente dañado que esté realizando una acción destructiva, como mover el código del componente receptor a una nueva ubicación de memoria antes de transferir el control).

El otro mecanismo provisto por JACAL para introducir restricciones sobre la combinación de los conectores y componentes es a través de la aridad asociada a los tipos de conectores.

3.1.3 ***Nivel de Comportamiento***

Los componentes se representan por sistemas de transición etiquetados. La etiqueta de una transición puede contener un nombre de port (punto de Input o de Output al que puede conectarse un conector), un tiempo mínimo y un tiempo máximo.

Como parte de la notación de JACAL original, si el nombre del port es primado -o sea: termina con un caracter prima (')-, se está denotando una respuesta (*return*), que es un retorno de control fluyendo a través de un conector en la dirección opuesta a la de éste.

Para cada port Input de un ***call***, ***interrupt*** o ***fork***, el componente debe contener un estado inicial unívoco (un nodo raíz) correspondiente. Ningún otro nodo en el componente puede ser el origen de una transición no-return asociada con ese conector. Los flujos de control recibidos comenzarán la ejecución en el estado inicial (en caso de ***call*** o ***fork***, esto es independiente de otros flujos que el componente eventualmente podría tener). Esto es así porque los componentes ***no pasivos*** no tienen un estado interno global (compartido por todos los flujos de control). Si el diseñador necesitara modelar un comportamiento de este tipo, necesitaría agregar un componente ***pasivo*** que sería accedido por transiciones del mencionado componente activo.

Las transiciones etiquetadas con nombres de port primados siempre deberían ser alcanzables desde estados iniciales correspondientes al mismo port, y no alcanzables desde estados iniciales correspondientes a otro flujo de control que transfieran los conectores.

JACAL no tiene una notación para datos enviados desde el origen de un conector a su destino. Distintos datos son representados por distintos conectores. Ej: si un componente A quiere hacer un llamado a la función argumento *calc* en el componente B, y éste se comportará distinto si el valor del argumento es mayor a 3 o menor o igual a 3, para modelar una situación en la que esos dos comportamientos distintos son relevantes para la arquitectura, deberían definirse dos conectores de tipo ***call*** entre A y B (podrían llamarse *calc>3* y *calc<=3*).

Se puede encontrar información adicional sobre JACAL original en [2].

3.2 ***Adaptación del lenguaje: construcción de JACAL 2.0***

Para poder analizar el paulatino progreso de nuestro trabajo (y con él, la evolución de muchas de las ideas que llevaron a transformar el JACAL original en JACAL 2.0), disponemos del ***Anexo II – Evolución del trabajo***, que acompaña y complementa el presente documento.

Respecto del producto final (el JACAL 2.0 y su entorno), a continuación se detallan algunas características de la nueva versión, analizadas en contraposición a su versión original, divididas temáticamente.

Especificación de la arquitectura

La nueva versión del lenguaje JACAL (cuya sintaxis genérica puede consultarse en el ***Anexo IV – Sintaxis JACAL 2.0 en BNF***) mantiene como objetivo el de ser un lenguaje de descripción de arquitecturas de software de propósito general, mediante el cual puedan representarse sistemas que adhieran a los diferentes estilos arquitecturales representativos del área, o bien a una combinación heterogénea de ellos.

La estructura global, correspondiente a la especificación de una arquitectura, se modifica de modo que contemple la definición de tres items fundamentales: componentes, conectores y configuración. De esta forma, se garantiza un mayor nivel de escalabilidad y reusabilidad, a la vez que se ajusta el lenguaje a un estándar de representación que permita ser considerado en un proceso de traducción a otro ADL mediante algún lenguaje de intercambio de arquitecturas.

En particular, el enfoque de diseño de la simulación de las arquitecturas de software con JACAL 2.0 se orientó a un esquema MVC (Model-View-Controller [20]), de modo de especificar por separado el modelo de datos, la interfaz con el usuario y la lógica de control, minimizando así el impacto de potenciales modificaciones en uno de dichos componentes respecto de los otros dos.

A continuación se detallan los cambios incorporados en esta nueva versión de JACAL (o sea, en JACAL 2.0 respecto del JACAL original):

Componentes

Se reclasifican los tipos de componente, de tal forma que su semántica no se limite a la descripción de su aridad (cantidad y tipos de ports asociados) sino que también deje abierta la posibilidad de completar la determinación del comportamiento por medio de la declaración explícita de transiciones entre estados internos provocados por interacciones definidas en su interfaz. Con estos criterios en mente, y de acuerdo con la bibliografía consultada, creímos conveniente basar esta clasificación en componentes reentrantes y repositorios.

Los componentes repositorios mantienen un flujo interno (token) activo entre invocaciones. Esta categoría es natural para la representación de espacios de almacenamiento pero también de los servicios/demonios o bien sesiones.

Los componentes reentrantes son desactivados una vez completada cada invocación.

Para hacer homogénea la declaración de la interfaz de los componentes y los conectores, se mantiene la clasificación de los ports en input/output, se agrega el port de tipo read (para asociar a componentes de tipo repositorio) y se incorpora la declaración de la capacidad de

recibir (input) o emitir (output) datos -tokens-. A su vez, se agrega un atributo que permite especificar las cantidades mínima y máxima de conectores asociables al port.

Se descarta la posibilidad de incorporar ports bidireccionales a efectos de hacer más claras tanto la especificación del comportamiento de un componente como la simulación de una arquitectura en tiempo de ejecución.

Conectores

Los conectores, al tener que definirse en forma independiente de los componentes, modifican su estructura de declaración.

Se incorpora el concepto de roles en un conector, bajo los que van a declararse las características de sus extremos, de una forma similar a lo expuesto para los ports de un componente.

Básicamente, un conector tendrá uno o más roles, dentro de los cuales se deben especificar las cantidades mínima y máxima de ports que se le podrán asociar.

Dentro de la configuración, estos roles deben ser instanciados con referencias a ports específicos de componentes, quedando de esta forma establecida la instancia de vinculación.

Como en la versión anterior, el lenguaje ofrece un conjunto de conectores predeterminados con una semántica implícita, que va a inducir el comportamiento global de la arquitectura en su combinación con el comportamiento de los componentes conectados.

Este conjunto de conectores está basado tanto en los conectores existentes en la versión anterior de JACAL, como en agregados en base a una detallada taxonomía de conectores hallada en la bibliografía, con el objetivo de contar con un dominio representativo de las distintas formas de comunicación, manteniendo de esta forma la idea de que se trate de un ADL de propósito general que abarque la mayor cantidad de estilos arquitecturales posibles.

En esta revisión de JACAL también consideramos la reducción del dominio original de conectores, en base a la unificación de aquéllos que a nuestro criterio podrían ofrecer un comportamiento similar o bien a la asimilación de algunos existentes por otros con nueva funcionalidad que los incluyen.

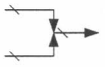
Tomamos, entonces, las siguientes decisiones con referencia al dominio de tipos de conectores:

- Mantener, de la versión anterior de JACAL, los ***datagram***, ***call***, ***fork***. ***read*** y ***write***.
- Renombrar el ***message*** por ***queue***, de forma que sea más claro intuir su comportamiento implícito.
- Eliminar los ***kill*** e ***interrupt***, dado que, si bien determinan una semántica particular en la comunicación entre *componentes*, ofrecen un menor nivel de abstracción que el resto.
- Agregar un nuevo *tipo de conector* ***arbitrator***, que permite sincronizar el acceso a un recurso compartido.

- Agregar los nuevos *tipos de conector* ***call_any***, ***queue_all*** y ***multi_write***, similar a los ***call queue*** y ***write***, respectivamente, pero de aridad 1:n.
- Agregar los *tipos de conector* ***publish*** y ***subscribe***, a los efectos de adaptar el mecanismo de eventos a este paradigma de comunicación ampliamente difundido en los productos comerciales.
- Agregar un nuevo *tipo de conector* ***rendez-vous***, que permite la sincronización del flujo de ejecución entre dos componentes.

A continuación, y a modo de resumen, brindamos una breve descripción acerca del comportamiento implícito ofrecido por cada conector predefinido en JACAL 2.0, junto con su nueva representación gráfica:

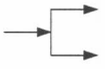
arbitrator → Es un nuevo tipo de conector que permite ofrecer a múltiples componentes el servicio de sincronización del acceso a un recurso compartido, administrándolo mediante una 'garantía de acceso' que se va alternando entre todos los clientes conectados.

Representación gráfica: 

call → Se mantiene la concepción original de la versión anterior: transfiere el control y espera la devolución del correspondiente resultado.

Representación gráfica:

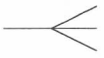
call_any → Similar al **call**, pero se puede invocar a más de un destinatario, y la respuesta puede venir de cualquiera de ellos; este comportamiento se describe con el no determinismo de las Redes de Petri (ver ***Anexo III – Semántica de Ports y Conectores***, que acompaña y complementa a este documento).

Representación gráfica: 

datagram → Se mantiene la concepción original de la versión anterior: si el receptor está esperando una comunicación a través de este conector, el mensaje es recibido; si no, se pierde. En cualquiera de los casos, el emisor continúa con su ejecución.

Representación gráfica:

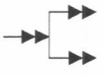
fork → Se mantiene la concepción original de la versión anterior: agrega un token en el receptor, el emisor continúa con su ejecución; aunque se agregó también la posibilidad de dividir en dos un flujo de control en el interior de un componente.

Representación gráfica: 

write → Se mantiene la concepción original del **write** de la versión anterior: cambia el estado del receptor, y el emisor continúa ejecutando.

Representación gráfica: 

multi write → Similar al **write**, pero se puede invocar a más de un destinatario, pudiéndose cambiar el estado a todos ellos.

Representación gráfica: 

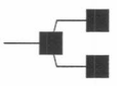
publish y subscribe → Se trata de nuevos tipos de conectores que permiten adherir al paradigma de comunicación Publish/Subscribe (ver cap 2.6).

Representación gráfica Publish:  y Subscribe: 

queue → Se mantiene la concepción original del **message** de la versión anterior: coloca el mensaje en una cola y continúa la ejecución del emisor.

Representación gráfica: 

queue all → Similar al **queue**, pero pudiendo dejar el mensaje en más de una cola.

Representación gráfica: 

read → Se mantiene la concepción original del **read** de la versión anterior: la ejecución en el emisor continúa de acuerdo con el estado del (único) receptor.

Representación gráfica: 

rendez-vous → Permite a un componente sincronizar su flujo de ejecución con el de un tercero.

Representación gráfica:

Configuración

La declaración de la configuración consiste en instanciar todos los conectores necesarios para obtener la arquitectura deseada. Para esto, a los roles de cada conector existente en el sistema hay que asociarles los ports de los componentes que van a estar vinculados con el conector, restringiendo esta acción mediante las características declaradas tanto en el conector como en el componente involucrado.

Nivel de comportamiento

A diferencia de JACAL original, durante una simulación de ejecución en JACAL 2.0, el usuario puede dinámicamente crear nuevos tokens haciendo doble click sobre los ports de input sin conectores asociados, y también puede especificar -para los llamados componentes *reentrantes*- que determinados estados tengan un *token inicial*, lo cual en la simulación se traducirá en un flujo de ejecución que no dependerá de la interacción con el usuario. Estos *tokens iniciales*, a la espera de un flujo de ejecución de los componentes *reentrantes*, en JACAL 2.0 se distinguen visualmente, representándoselos de color blanco.

Vale resaltar que uno de los objetivos que nos propusimos explícitamente para mejorar respecto del JACAL original fue el tratar de lograr llevar a la práctica el funcionamiento pretendido para los componentes *reentrantes* (o sea: que empiecen con un flujo de ejecución sin que haga falta inicializarlos), y que se logró merced al agregado de los *tokens iniciales* y a la definición de comportamiento y semántica para los ports (ver ***Anexo III – Semántica de Ports y Conectores***, que acompaña y complementa a este documento).

También se introdujo la noción de *transiciones fork*, que dividen en dos un flujo en el interior de un componente (quedando visualmente el flujo como una Y), y que duplican tokens; de este modo, es posible representar situaciones como la de un componente que viene ejecutando, decide ponerse a escuchar en un port –ej: de una cola o de un *datagram*- y además sigue su ejecución.

3.3 Herramienta JACAL 2.0

En esta sección explicamos el funcionamiento de JACAL 2.0, la herramienta de software que implementamos, diseñamos y desarrollamos desde cero -aunque tomando como base conceptual al JACAL original (ver [2])-, y que sirve como solución de MS Visio para editar arquitecturas en ella y simular su ejecución.

Software requerido

Para poder utilizar JACAL 2.0, se deberá contar mínimamente con el siguiente software instalado:

- Microsoft Visio 2002
- Jacal.vst: Template del lenguaje
- Jacal.vss: Stencil del lenguaje

Exportación

Mediante la opción *Save / Save As*, se dispara el proceso de exportación, el cual concatena sobre el final del archivo exportado el esquema JACAL XML correspondiente a la arquitectura a preservar (ver ***Anexo I – Código XML generado***).

Validación

Una vez presionado el botón *Validar Arquitectura* se dispara el proceso de comprobación de su consistencia. El mismo incluye reglas de validación para todos los componentes de la arquitectura, tanto en el nivel de interfaz como en el de comportamiento, y se informan –en caso de existir- todos los errores en un recuadro emitido a tal efecto.

Simulación

I. Una vez presionado el botón "Simular Arquitectura", se inicia el proceso de simulación. Como primera etapa de la misma, se invoca a la rutina de *Validación* de la arquitectura, con el fin de comprobar que la arquitectura a simular sea válida.

II. En caso que la arquitectura sea válida, se inicia su simulación. Se configuran los posibles tokens iniciales y se permite al usuario crear nuevos tokens haciendo doble click sobre los ports de input sin conectores asociados.

III. El proceso de *Simulación* basa su funcionamiento tanto en el comportamiento explícito modelado por el usuario como en el comportamiento implícito inherente a cada tipo de conector incluido en el lenguaje.

IV. Los tokens creados por el usuario se van configurando con distintos colores, para facilitar su seguimiento.

V. Los tokens correspondientes a componentes de tipo *repositorio* se grafican con color negro.

VI. Los tokens iniciales y/o a la espera de un flujo de ejecución de los componentes *reentrantes* se grafican con color blanco.

VII. El proceso de *Simulación* genera automáticamente, en el mismo directorio de la arquitectura en ejecución, un archivo de log en formato XML, donde se registra toda la información relacionada a cada iteración instanciada, facilitando con esto la posible automatización del análisis de diferentes aspectos dinámicos de la arquitectura definida.

VIII. Mediante el botón 'Configurar velocidad', es posible seleccionar el delay entre iteraciones, de modo de revisar más detenidamente cada paso de la simulación.

IX. Para detener el proceso de simulación, deberá presionarse el botón *Detener Simulación*. El proceso de *Simulación* se detendrá automáticamente si en alguna iteración se produce alguna excepción. De ser así, se informa el número de la iteración hasta la cual se llegó y la causa de la excepción.

3.4 Comparación con otros ADLs

De acuerdo con lo comentado a lo largo de la sección 2.2 y habiendo detallado las características de JACAL 2.0 en las secciones precedentes, nos parece importante proponer una comparación entre la herramienta que hemos desarrollado y algunos lenguajes existentes en el mercado que han sido considerados como ADLs, utilizando como base el exhaustivo análisis realizado en [28], con lo cual no se incluyeron lenguajes como Promela ([25]), ArTek o UML:

3.4.1 Acme

Se trata, más que de un ADL propiamente dicho, de un lenguaje de modelado de especificaciones arquitecturales entre diferentes ADLs, diseñado originalmente con la idea de "proveer un lenguaje común que pudiera usarse para soportar el intercambio de descripciones arquitecturales entre una variedad de herramientas de diseño arquitectural" [29]. Esta diferencia de objetivos entre ambas herramientas dificulta su comparación (y la hace un tanto estéril), aunque sí es posible resaltar algunos aspectos.

JACAL fue originalmente ideado como un ADL, mientras que Acme fue pensado como un lenguaje de mayor nivel de abstracción, ya que su principal objeto de representación pretendían ser los mismos ADLs.

JACAL 2.0 especifica la semántica de los ports con Redes de Petri Coloreadas, determinando su comportamiento; en Acme sólo se especifica la semántica de los componentes a modo de lista de propiedades, y no se interpretan sino como documentación.

Asimismo, una especificación en JACAL 2.0 permite generar código XML, mientras que Acme está pensado como una herramienta de modelado, sin soporte de generación de código.

La simulación de una arquitectura es posible en JACAL 2.0 (permitiendo un eficiente manejo de fallas y la detección anticipada de propiedades no deseadas), pero no en Acme.

Coherentemente con los objetivos propuestos al momento de su creación –y a diferencia de JACAL-, Acme ya posee una biblioteca predefinida (*AcmeLib*, cuyo código está disponible en C++ y en Java) con una serie de clases que permiten manipular una representación Acme en cualquier aplicación Microsoft o framework .net.

3.4.2 MetaH AADL

MetaH AADL ([30]) es un set de herramientas ideado para el modelado de arquitecturas, enfocado originalmente hacia aplicaciones de Guía, Navegación y Control (GN&C) y de diseño aeronáutico; está ligado exclusivamente a desarrollos hechos en Ada en el dominio de referencia, y entre su kit de herramientas se cuenta un editor gráfico que permite trabajar en entornos Windows.

AADL es un ADL orientado a la especificación de sistemas embebidos de seguridad crítica y tiempo real, creado como parte del kit de MetaH.

En este sentido es que también podemos decir que difieren los objetivos propuestos inicialmente para el diseño de la herramienta respecto de JACAL 2.0, que fue ideado para funcionar como un ADL de propósito general (énfasis de MetaH en monitoreos de performance o seguridad, que JACAL no posee, al menos hasta hoy).

3.4.3 Rapide

Como JACAL, la idea detrás de Rapide es la pertenencia a la generación de lenguajes conocida como EADLs (Executable Architecture Definition Languages), de propósito general. Sin embargo, la estructura de Rapide es mucho más compleja (y más difícil de manejar, pro más que su modelo de ejecución sea único) que la de JACAL 2.0, al punto de tener que articular 5 lenguajes diferentes para definir una arquitectura: uno para describir las interfaces de los componentes; otro para describir el flujo de eventos entre componentes; un tercero para especificar las restricciones sobre el comportamiento de los componentes; otro para describir los módulos ejecutables; y un último para describir patrones de los eventos. En cambio, con la semántica de Redes de Petri coloreadas para especificar el comportamiento de ports y conectores y el soporte en la muy difundida aplicación MS Visio, JACAL 2.0 se hace mucho más simple de manipular, y proporciona a la vez facilidades de manejo de fallas, posibilidad de especificar (y simular) la administración de recursos compartidos, configuración dinámica (publish / subscribe), detección de propiedades no deseadas, etc.

Rapide introduce un modelo de ejecución basada en eventos para sistemas distribuidos y (a diferencia de JACAL) sensibles al tiempo: el modelo de *posets* por tiempo. Los *posets* proveen una base formal detallada para la construcción de herramientas de prototipación de

distintos ciclos de vida en el tiempo, apuntando al análisis de performance (que JACAL no tiene) y correctitud de sistemas distribuidos sensibles al tiempo.

3.4.4 UniCon

Es un ADL organizado alrededor de dos construcciones simétricas: componentes y conectores ([31]), y fue creado con el propósito de generar código ejecutable a partir de una descripción, a partir de los componentes primitivos adecuados. Proporciona una herramienta de diseño para construir configuraciones ejecutables basadas en tipos de componentes, implementaciones y algunos tipos de conectores, y permite el manejo de métodos de análisis de tiempo real (a diferencia de JACAL), pero no posee un fuerte soporte semántico (como el que tiene JACAL a través de las Redes de Petri coloreadas, que le permiten especificar el comportamiento de ports y conectores) sino a través de una lista de propiedades.

Si bien UniCon genera código, lo hace en lenguaje C, una elección que no parece muy adecuada para el soporte de este tipo de codificaciones, tratándose de un lenguaje de relativo bajo nivel, a diferencia del XML en el que se traduce una arquitectura JACAL 2.0 (o incluso en el que directamente se puede construir o escribir una arquitectura – ver documento ***Anexo I – Código XML generado***, que acompaña y complementa esta presentación).

3.4.5 Wright

El propósito del desarrollo de Wright fue proveer una base formal para describir y analizar arquitecturas de software, introduciendo soportes para especificación formal de nuevos tipos de conexión arquitectural, para definiciones formales de estilos arquitecturales y para reglas de chequeo de consistencia y completitud de diseños arquitecturales ([25]).

En Wright se declaran restricciones declaradas por un estilo a modo de predicados de primer orden que deben ser satisfechos por cualquier configuración de la que se declare que es miembro de ese estilo (de alguna manera similar a lo que sería JACAL 2.0 si tuviera model-checking, propiedad que hoy no tiene y que está incluida en el capítulo ***7 – Posibles extensiones***), aunque vale aclarar que no es posible verificar la conformidad de una configuración a un estilo estándar.

Si bien Wright tiene soporte de modelo semántico (CSP – Communicating Sequential Process), permitiendo así analizar -por ejemplo- los conectores para verificar la inexistencia de deadlocks, estas verificaciones se aplican estáticamente, y no sobre simulación alguna (como sí hace JACAL 2.0), si bien al existir soluciones que pueden tratar código CSP en plataforma Windows es posible visualizar el modelo diseñado o chequear su consistencia.

Por otra parte, Wright no proporciona notación gráfica ni genera código (ya que se considera a sí mismo una notación de modelo autocontenida); JACAL 2.0, como ya se explicó, permite exportar su código a notación XML, un estándar en lenguajes de especificación.

4 CASOS DE ESTUDIO

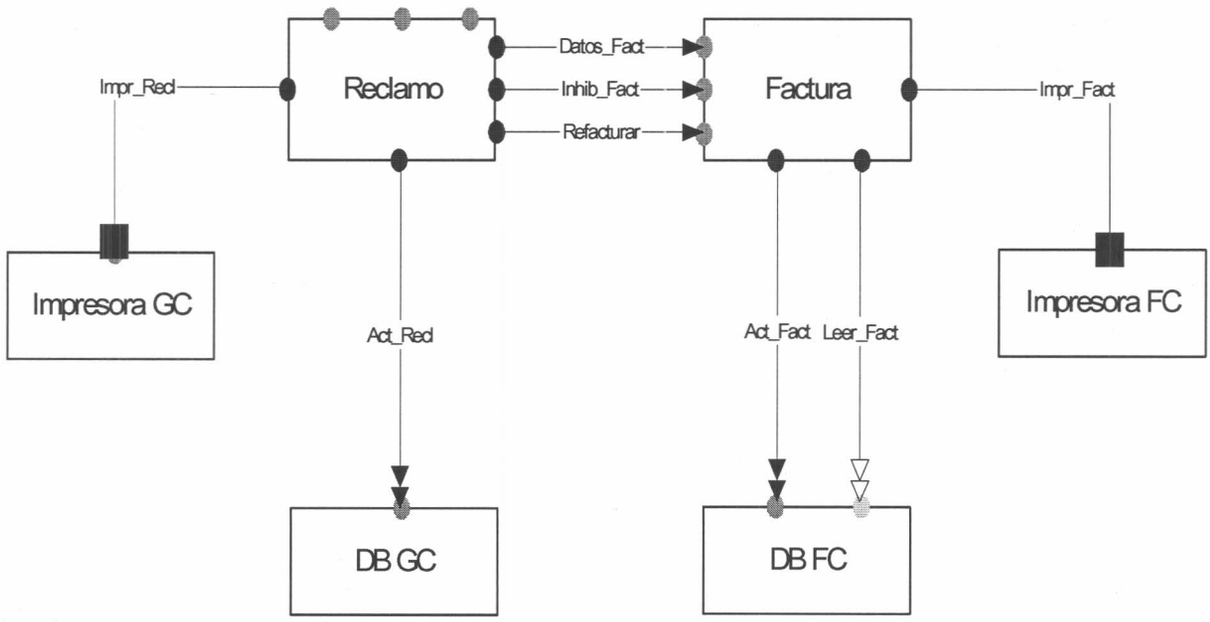
Para explotar la variedad de conectores que ofrece el área de arquitecturas de software y demostrar la potencialidad de JACAL 2.0, se muestra una serie de Casos de Estudio en los que se tratarán de expresar facilidades de comunicación y coordinación mediante los conectores, así como funcionalidad para abarcar la técnica publish/subcribe (eventos), independencia del comportamiento de los componentes de la interfaz, y componentes que ofrezcan servicios que no necesiten inicialización interactiva.

CASO 1: Módulo de Reclamos de Facturación de tráfico internacional telefónico

El sistema modela una simplificación funcional de una aplicación de reclamos de facturación de tráfico internacional (basada en la experiencia profesional de dos de los autores de este trabajo), mediante el cual los clientes pueden ingresar pedidos de refacturación o reintegros por cargos mal facturados.

Para resolver la gestión de un reclamo de este tipo, participan básicamente dos sistemas heterogéneos: Gestión Comercial (representado por el componente *Reclamo*) y Facturación & Cobros (representado por el componente *Factura*).

A continuación, se muestra el nivel de interfaz de la arquitectura propuesta:



Se diseñan 2 operaciones básicas del ciclo de vida de un reclamo: Ingreso y Dictamen.

Ingreso

A partir del llamado de un cliente, el empleado del Call Center efectúa el ingreso del reclamo en la aplicación de Gestión Comercial (uno de los tres puntos de entrada disponibles en el componente *Reclamo*), necesitando para esto extraer datos referentes a la factura del cliente en cuestión (conector *Datos_Fact*, de tipo **Call**). Este servicio es provisto por un

procedimiento de la aplicación Facturación & Cobros (conector *Leer_Fact*, de tipo **Read**), el cual recupera la información de la Base de Datos local (componente *DB FC*). Una vez especificado el detalle del reclamo, el mismo es confirmado (conector *Act_Recl*, de tipo **Write**), momento en el cual se dispara un proceso de inhibición en la aplicación F&C con el fin que el cliente no sea contemplado en los procesos de incomunicación por morosidad (para esto, se ejecutarán en forma secuencial el conector *Inhib_fact*, de tipo **Call**, y el conector *Act_Fact*, de tipo **Write**).

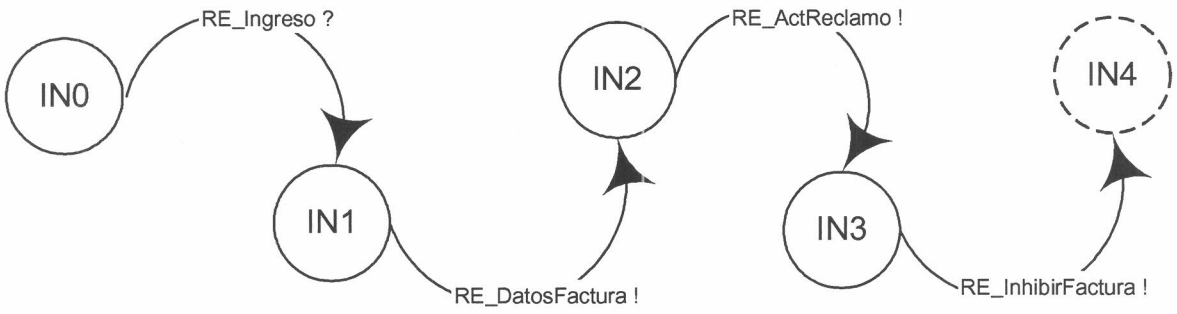
Dictamen

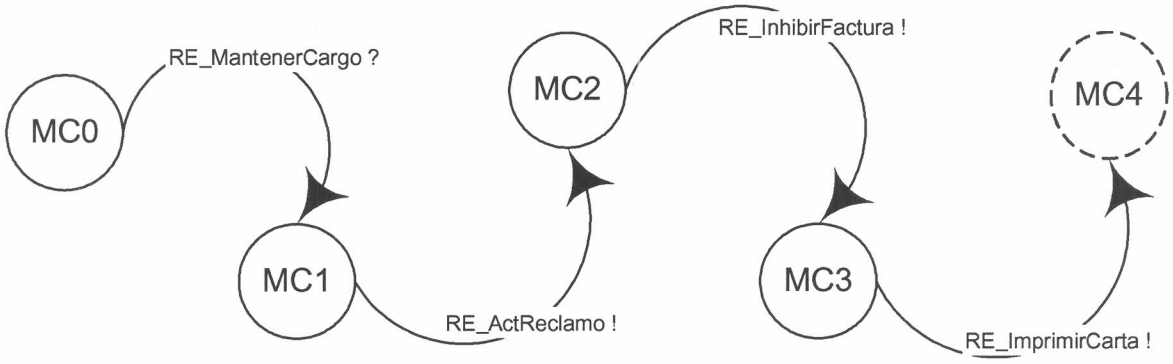
Los distintos "back offices" de la empresa analizan los reclamos de facturación ingresados y determinan si corresponde o no realizar una devolución al cliente. La entrada al sistema de esta determinación se representa como los otros dos puntos de entrada disponibles en el componente *Reclamo*.

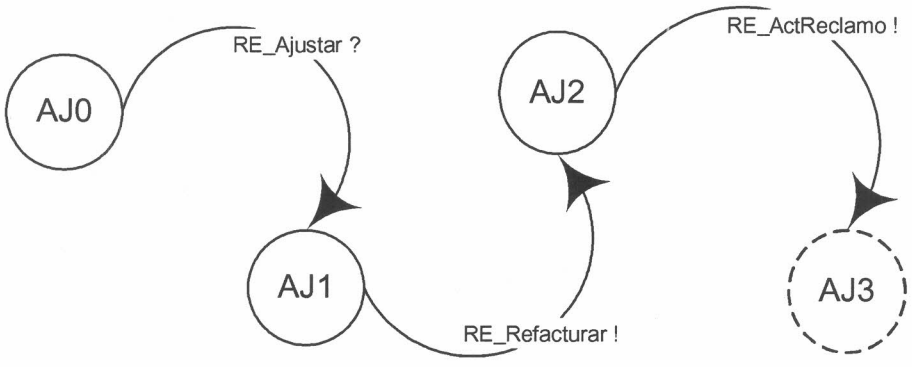
Si la decisión es no dar lugar al reclamo del cliente, se dictamina el cierre del mismo en la aplicación de gestión comercial (conector *Act_Recl*, de tipo **Write**), lanzando el proceso de desinhibición del cliente en la aplicación F&C para que vuelva a ser considerado en los procesos de incomunicación por morosidad (para esto, y al igual que en la transacción de **Ingreso**, se ejecutarán en forma secuencial el conector *Inhib_fact*, de tipo **Call**, y el conector *Act_Fact*, de tipo **Write**). A su vez, se imprime una carta de notificación al cliente de la resolución (conector *Impr_Recl*, de tipo **Queue**).

Si la decisión es dar lugar al reclamo del cliente, se lanza una refacturación de los cargos ajustados en la aplicación F&C (para esto, se ejecutarán en forma secuencial el conector *Refacturar*, de tipo **Call**, y el conector *Act_Fact*, de tipo **Write**). A su vez, se imprime la nueva factura (conector *Impr_Fact*, de tipo **Queue**) y se procede al cierre del reclamo en la aplicación de Gestión Comercial (conector *Act_Recl*, de tipo **Write**).

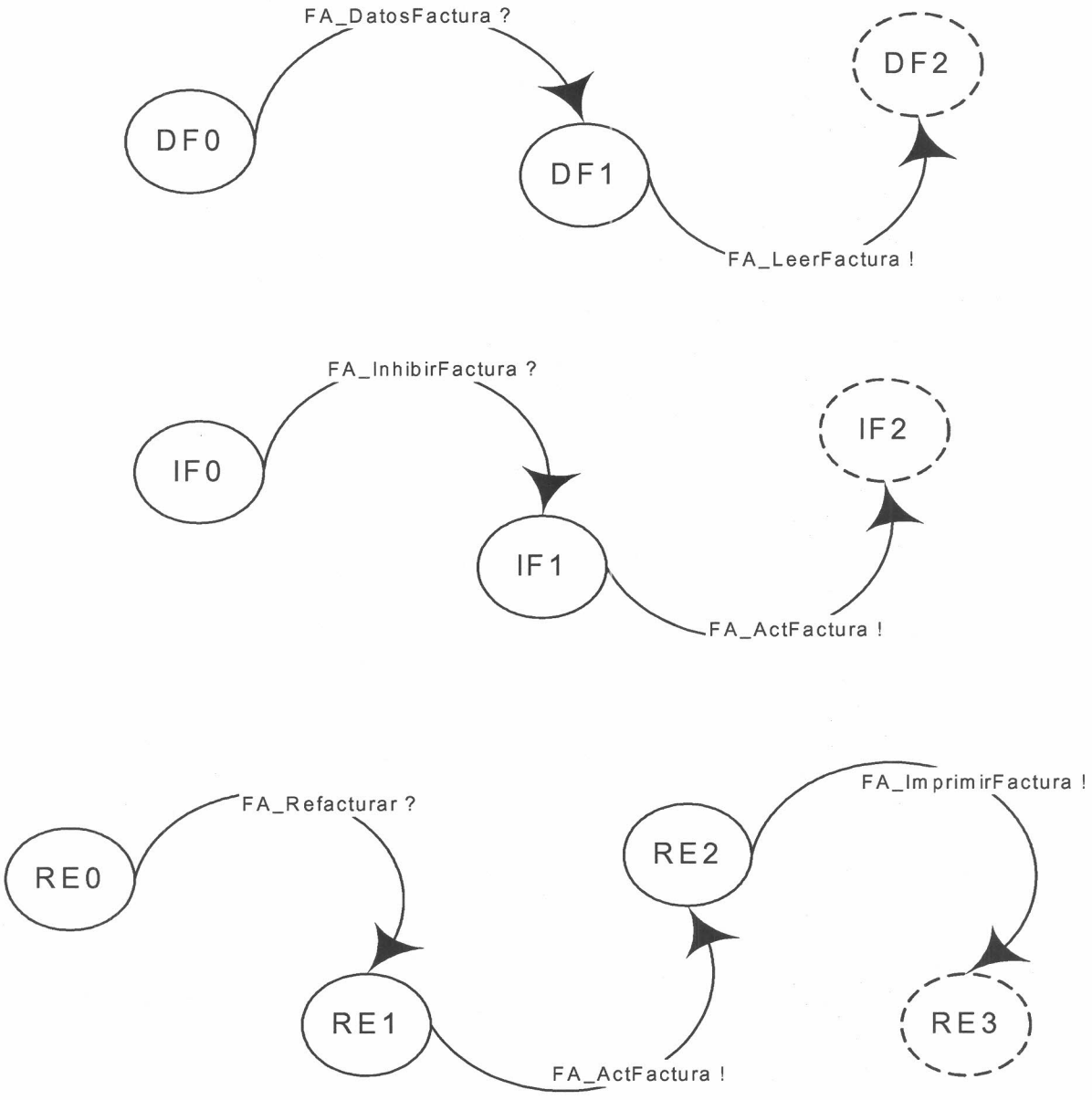
A continuación, se muestra el nivel de comportamiento definido para los componentes *Reclamo* y *Factura*, donde se modela la funcionalidad especificada anteriormente:







(Nivel de comportamiento de componente *Reclamo*)



(Nivel de comportamiento de componente *Factura*)

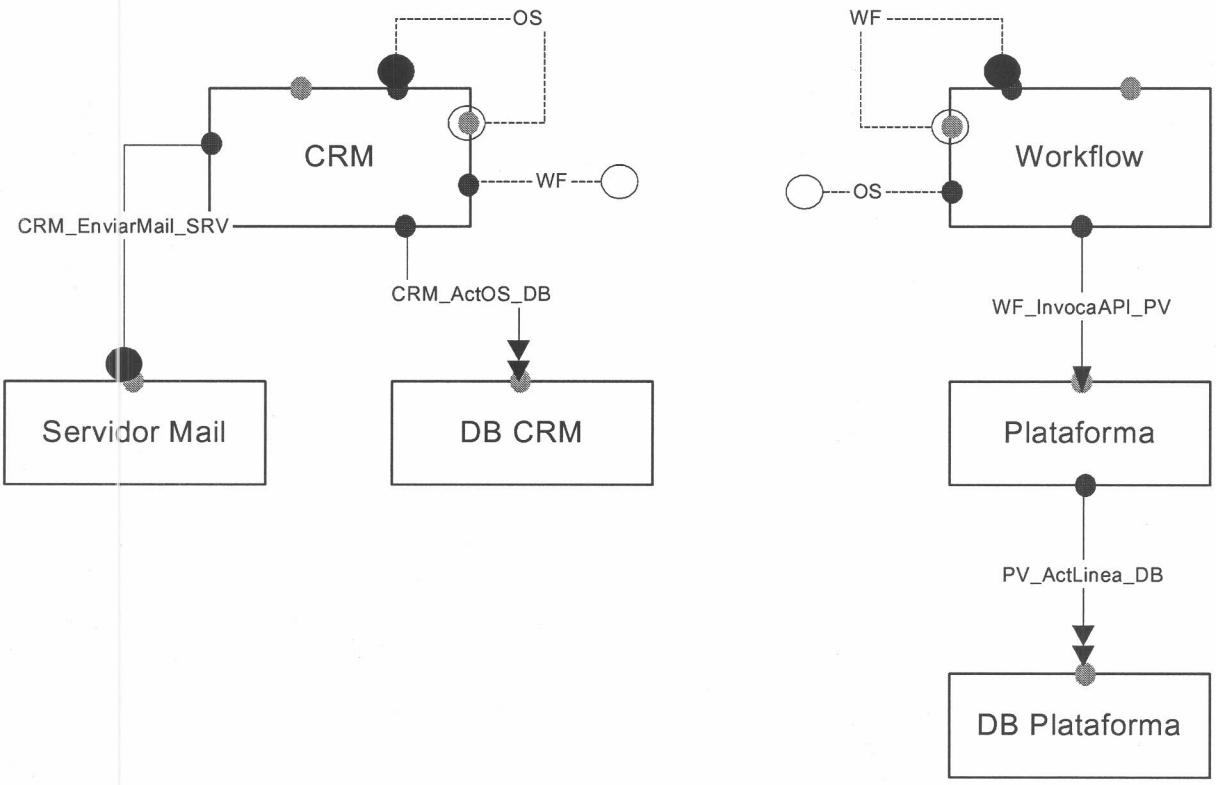
El código XML completo que se genera a partir de la exportación de este Caso se expone en el documento **Anexo I – Código XML generado**, que acompaña y complementa esta presentación.

CASO 2: Alta de servicio en plataforma de voz

El sistema modela una simplificación funcional del alta de servicios en una plataforma de voz, para lo cual deben interactuar la aplicación CRM y un módulo de provisión que ofrece como servicio la instanciación y ejecución de un workflow mediante el cual se gestiona la aplicación de

los comandos de activación sobre la plataforma. Una característica a destacar de la arquitectura es que el CRM y la solución de provisión se encuentran desacoplados y el mecanismo de interacción entre ambos es mediante el intercambio de mensajes asincrónicos.

A continuación, se muestra el nivel de interfaz de la arquitectura propuesta:



Se diseña el ciclo de vida del alta de una línea sobre la plataforma:

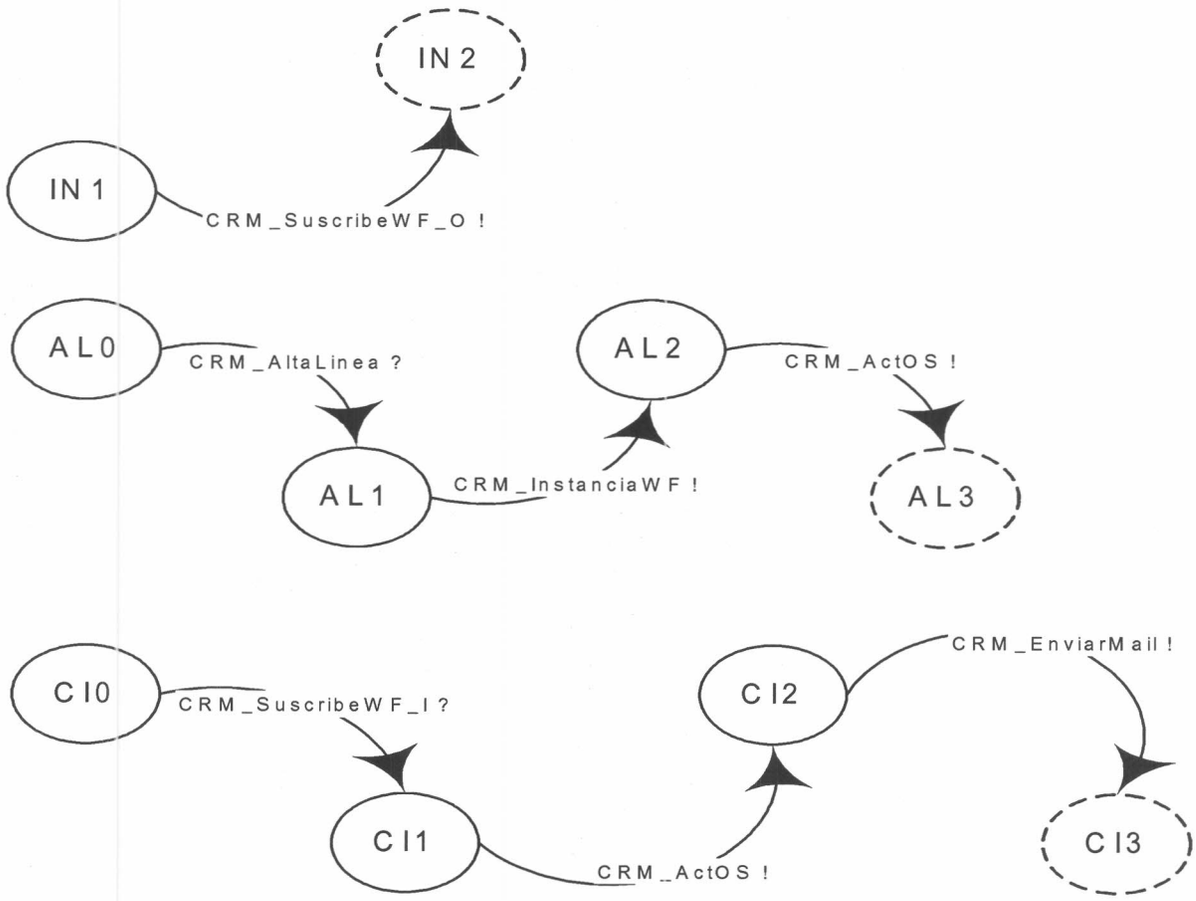
Se definen tokens iniciales sobre la arquitectura, de forma tal que la aplicación CRM suscriba al tema a publicar por el workflow y viceversa (conectores *OS* y *WF*, de tipo **Subscribe**).

Una vez ingresado el pedido de alta sobre el CRM (punto de entrada disponible en el componente *CRM*), mediante la publicación de un mensaje, el mismo invoca en forma implícita a la solución de provisión con el objetivo de que se instancie el workflow correspondiente (conector *WF*, de tipo **Publish**, y conector *CRM_ActOS_DB*, de tipo **Write**). Por cada workflow de alta a cumplir, el mismo invoca a las APIs ofrecidas por la plataforma de voz mediante las cuales se terminan aplicando los comandos sobre la Base de Datos (conector *WF_InvocaAPI_PV*, de tipo **Call**, y conector *PV_ActLinea_DB*, de tipo **Write**).

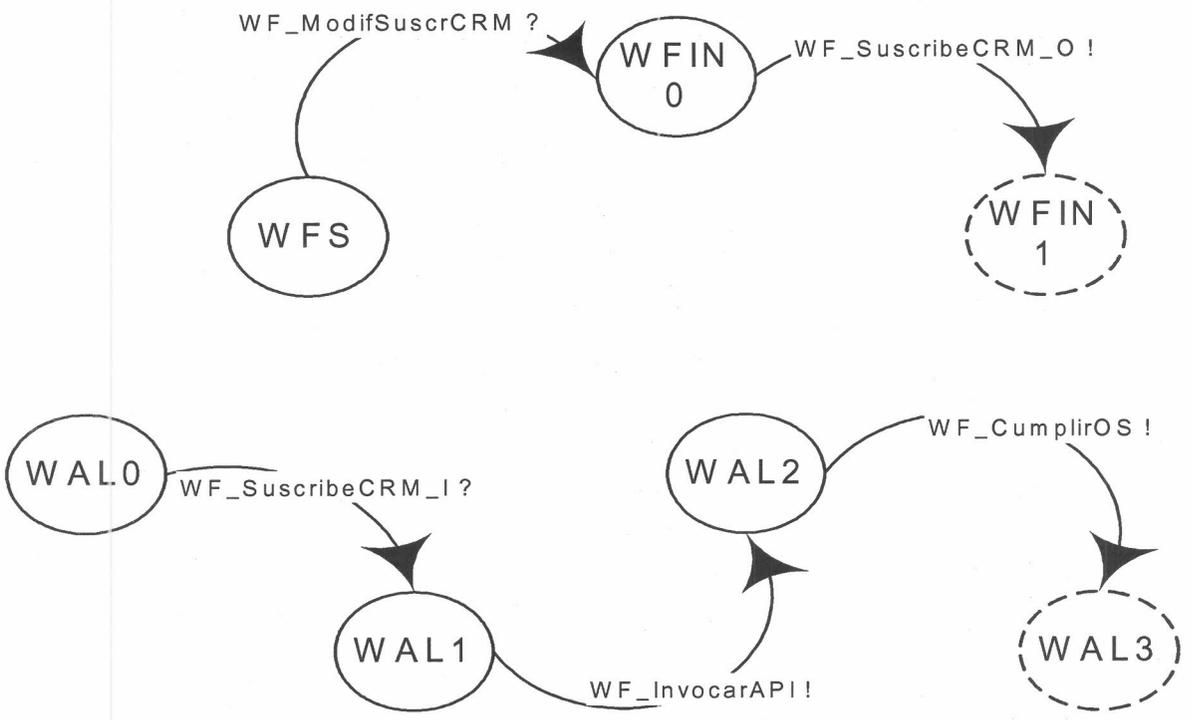
Una vez cubierta la transacción de alta sobre la plataforma, desde el workflow se invoca en forma implícita al CRM para que se concluya la orden de servicio (conector *OS*, de tipo **Publish**), enviando en esta instancia un mail a la casilla del supervisor de la oficina comercial (conector *CRM_EnviarMail_SRV*, de tipo **Datagram**).

En el componente *Workflow* se define un punto de entrada, mediante el cual se podrá invertir la condición de suscripción al tema a publicar por el CRM, con el objetivo de comprobar la capacidad de configuración dinámica de una arquitectura que implementa el mecanismo de comunicación publish/subscribe.

A continuación, se muestra el nivel de comportamiento definido para los componentes *CRM* y *Workflow*, donde se modela la funcionalidad especificada anteriormente:



(Nivel de comportamiento de componente *CRM*)



(Nivel de comportamiento de componente *Workflow*)

En el CD que adjuntamos con el presente documento se dispone de una serie de archivos de MS Visio que contienen distintos Casos de Estudio preparados, donde los primeros 2 son los detallados en esta Sección, y el resto (del 3 al 9) termina de mostrar el funcionamiento y la representación del resto de los conectores y transiciones de JACAL 2.0.

5 CONCLUSIONES

Así como la creación de JACAL introdujo la noción de que un lenguaje arquitectural podía proveer descripciones lo suficientemente completas como para poder ejecutar arquitecturas de software, con esta Tesis no sólo estamos reforzando esa idea (a través de importantes modificaciones a la herramienta en sí, incorporándole una serie de nuevas características y mejorando u optimizando otras ya existentes) sino que también estamos extendiendo las posibilidades de representación de una arquitectura que puede probarse antes de su implementación, al lograr una correlación directa entre la especificación de los modelos y su representación gráfica, utilizando para ello uno de los entornos de diseño más difundidos en el mercado.

6 CONTRIBUCIONES

Se logró enriquecer notablemente el concepto subyacente en una herramienta ya existente (JACAL) por medio de un desarrollo realizado *desde cero*. Esta nueva herramienta (JACAL 2.0) permite importar código desde XML (generando un esquema arquitectural), y también exportar el esquema de la arquitectura hacia XML, logrando así una interacción dinámica entre el lenguaje XML y la representación gráfica de una arquitectura en MS Visio.

Asimismo, también es posible ahora validar si una arquitectura JACAL XML plasmada en el MS Visio es consistente en términos de JACAL 2.0; y si no lo es, cuáles son todos los errores de diseño existentes (tanto en la hoja de Interfaz como en las de Comportamiento).

Vale destacar también el hecho de que los conectores predefinidos en JACAL 2.0 y sus funcionalidades cubren gran parte de la taxonomía actual de los conectores existentes en la literatura investigada.

También se hace necesario subrayar el encapsulamiento del Nivel de Interfaz respecto del Nivel de Diseño. De este modo, en JACAL 2.0 se puede definir el comportamiento de un componente independientemente de los conectores que se asocien a sus ports y, por lo tanto, de la configuración de las arquitecturas en las que esté involucrado.

JACAL 2.0 también permite especificar, para los llamados componentes *reentrantes*, que determinados estados tengan un token inicial, lo cual en la simulación se traducirá en un flujo de ejecución que no dependerá de la interacción con el usuario.

También se redefinió el comportamiento de eventos en el JACAL original, asemejando este concepto a la técnica *publish/subscribe* utilizada en el middleware actual y combinando de esta forma la invocación implícita entre componentes junto a la posibilidad de modificar en forma dinámica la configuración de la arquitectura.

Finalmente, vale destacar la incorporación del concepto de sincronización en la arquitectura (conector *rendez-vous*), y de un grupo de conectores que permite diseñar una mayor variedad de operaciones multicast.

7 POSIBLES EXTENSIONES

- Incorporación de *model-checking* (técnica de verificación automática que, dado un modelo del sistema y una fórmula, comprueba si el modelo satisface la fórmula).
- Inclusión de nociones temporales, como eventos *publish / subscribe* por *leasing* (en los que el suscriptor podría especificar períodos de tiempo durante los cuales se suscribiría a las clases de evento por las que registraría interés), o comportamientos de cualquier conector

dependientes de alguna referencia temporal (ej: especificar que ante la falta de respuesta a un *call* por determinado tiempo, se ejecute alguna otra acción alternativa).

- Definición de nuevos conectores por parte del usuario.
- Asistencia al usuario ante errores de diseño sugiriendo posibles correcciones.
- Asistencia al usuario ante errores de diseño destacando los objetos involucrados en el error al marcar o hacer doble click sobre la línea de error correspondiente.
- Adaptabilidad de la arquitectura (y de la simulación) a cambios dinámicos de configuración.
- Permitir, en la especificación del comportamiento de un componente, hacer referencia a una arquitectura almacenada en un archivo externo o bien a un comportamiento estándar debidamente identificado en el esquema.

8 REFERENCIAS

[1] Software Engineering Institute (Carnegie Mellon University), "Software Technology Roadmap", 2005

[2] N.Kicillof, "Running Software Architectures", Tesis de Licenciatura (Dir: Dr. Daniel Yankelevich), 1997

[3] N.Kicillof, D.Yankelevich, "Detecting and solving architectural problems with JACAL", First Australian Workshop on Constructing Software Engineering Tools (AWCSET 1998)

[4] D.Garlan, M.Shaw, "An introduction to software architecture", School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, January 1994

[5] N.Medvidovic, R.Taylor, "A classification and comparison framework for software architecture description languages", IEEE Transactions on Software Engineering, Volume 26, Issue 1, 2000

[6] N.R.Mehta, N.Medvidovic, S.Phadke, "Towards a taxonomy of software connectors", 22nd International Conference on Software Engineering, Limerick, Ireland, 2000

[7] D.Bakken, "Middleware", http://www.eecs.wsu.edu/~bakken/middleware-article-bakken.pdf

[8] J.Angel, "Lesson 134: Middleware", Network Magazine, 05/01/2001

[9] Talarian Corporation, "Everything You need to know about Middleware: Mission-Critical Interprocess Communication", http://www.talarian.com/, 1999

[10] Advanced Quality Solutions, "XML Roadmap", www.aqs.es, 2001

[11] A.Reino Romero, "Introducción a XML" versión 2.0, http://sunsite.unam.mx/archivos/xml/IntroXMLc.pdf, 26/01/2000

[12] G.Muhl, "Large Scale Content-Based Publish/Subscribe Systems", PhD thesis, Technische Universitat Darmstadt, 2002

[13] G.Cugola, D.Frey, A.Murphy, G.P.Picco, "Bringing dynamic reconfiguration into Publish-Subscribe Systems", Dipartimento di Elettronica e Informazione, Politecnico di Milano

[14] D.Luckham, J.Kenney, L.Augustin, J.Vera, D.Bryan, W.Mann, "Specification and analysis of system architecture using Rapide", IEEE Transactions on Software Engineering, Volume 21, Issue 4 (April 1995)

[15] R.N.Taylor, N.Medvidovic, K.M.Anderson, E.J.Whitehead Jr., J.E.Robbins, K.A.Nies, P.Oreizy and D.L.Dubrow, "A Component- and message-based Architectural Style for GUI Software", IEEE Transactions on Software Engineering, Volume 22, Issue 6 (June 1996)

[16] P.T.Eugster, R.Guerraoui, J.Sventek, "Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction", Technical Report DSC/2000/003, Ecole Polytechnique federale de Lausanne (EPFL), Communication Systems Department (DSC), Switzerland, 2000

[17] E.Dashofy, N.Medvidovic, R.Taylor, "Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures", 21st International Conference on Software Engineering, Los Angeles, USA, 1999

[18] D.Helic, N.Scherbakov, "Introduction to Software Architecture", http://coronet.iicm.edu/wbtmaster/courses/swp5_startc.htm

[19] S.Pruitt, D.Stuart, W.Sull, T.W.Cook, "The Merit of XML as an Architecture Description Language Meta-Language", MCC (Microelectronics and Computer Technology Corp.), Austin, Texas, USA, 1998

[20] S.Burbeck, "Applications Programming in Smalltalk-80®: How to use Model-View-Controller (MVC)", http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[21] W.van der Aalst, "High level Petri nets – Extending classical Petri nets with color, time and hierarchy", University of Technology, Faculty of Technology Management, Department of Information and Technology, Eindhoven, The Netherlands.

[22] K.Jensen, "A Brief Introduction to Coloured Petri Nets", Computer Sciences Department, University of Aarhus, Denmark, 1996

[23] S.Christensen, K.H.Mortensen, "Parametrisation of Coloured Petri Nets", Computer Sciences Department, University of Aarhus, Denmark, 1997

[24] G.J.Holzmann, "Basic SPIN Manual", Technical report, Bell Laboratories, 1994

[25] R.Allen, D.Garlan, "The Wright architectural specification language", Carnegie Mellon University, School of Computer Science, Pittsburgh, September 1996

[26] W.Reisig, "Petri nets – An introduction", EATCS Monographs on Theoretical Computer Science, volume 4. Springler-Verlag, 1985

[27] L.Bass, P.Clements, R.Kazman, "Software Architecture in Practice", Addison Wesley Professional, 2003

[28] C.Reynoso, N.Kicillof, "De Lenguajes de descripción arquitectónica de Software (ADL)", 2004

[29] ABLE Group, "The *Acme* Architectural Description Language", School of Computer Science, Carnegie Mellon University, 1998

[30] "The MetaH AADL Toolset – A better way to build hard, real-time, safety critical embedded systems", http://www.honeywell.com/sites/docs/DKRHVMR134XK8DQRHFA5JVOVUZQ23ENOK.pdf

[31] G.Zelesnik, "The UniCon Language Reference Manual", School of Computer Science, Carnegie Mellon University, 1996

Tesis de Licenciatura

Definición e Implementación de un Lenguaje de Descripción de Arquitecturas de Software

Anexo I – Código XML generado

En el presente Anexo se muestra un ejemplo de código XML generado a partir de la exportación de la representación de una arquitectura determinada en JACAL 2.0; en particular se muestra aquí el código derivado de la exportación del ***Caso de Estudio 1*** detallado en el Informe de Tesis (cuerpo principal, sección **4 Casos de Estudio**):

```
<SolutionXML Name='JACAL'>
   <architecture name='Caso 1'>
      <components>
         <component name='Reclamo' type='reentrant'>
            <interface>
               <port name='RE_Ingreso' type='input' min='0' max='0' />
               <port name='RE_MantenerCargo' type='input' min='0' max='0' />
               <port name='RE_DatosFactura' type='output' min='1' max='1' />
               <port name='RE_ActReclamo' type='output' min='1' max='1' />
               <port name='RE_InhibirFactura' type='output' min='1' max='1' />
               <port name='RE_Ajustar' type='input' min='0' max='0' />
               <port name='RE_ImprimirCarta' type='output' min='1' max='1' />
               <port name='RE_Refacturar' type='output' min='1' max='1' />
            </interface>
            <behavior>
               <state name='IN1' token_ini='0' entry_port='' end_state='FALSE' />
               <state name='IN2' token_ini='0' entry_port='' end_state='FALSE' />
               <state name='IN3' token_ini='0' entry_port='' end_state='FALSE' />
               <state name='IN4' token_ini='0' entry_port='' end_state='TRUE' />
               <state name='IN0' token_ini='0' entry_port='RE_Ingreso' end_state='FALSE' />
               <state name='MC0' token_ini='0' entry_port='RE_MantenerCargo' end_state='FALSE' />
               <state name='MC1' token_ini='0' entry_port='' end_state='FALSE' />
               <state name='MC2' token_ini='0' entry_port='' end_state='FALSE' />
               <state name='MC3' token_ini='0' entry_port='' end_state='FALSE' />
               <state name='MC4' token_ini='0' entry_port='' end_state='TRUE' />
               <state name='AJ0' token_ini='0' entry_port='RE_Ajustar' end_state='FALSE' />
               <state name='AJ1' token_ini='0' entry_port='' end_state='FALSE' />
               <state name='AJ2' token_ini='0' entry_port='' end_state='FALSE' />
               <state name='AJ3' token_ini='0' entry_port='' end_state='TRUE' />
               <transition trans_from='IN1' trans_port='RE_DatosFactura' trans_value=''>
                  <trans_to state = 'IN2' />
               </transition>
               <transition trans_from='IN2' trans_port='RE_ActReclamo' trans_value=''>
                  <trans_to state = 'IN3' />
               </transition>
               <transition trans_from='IN3' trans_port='RE_InhibirFactura' trans_value=''>
                  <trans_to state = 'IF2' />
               </transition>
               <transition trans_from='IN0' trans_port='RE_Ingreso' trans_value=''>
                  <trans_to state = 'IN1' />
               </transition>
               <transition trans_from='MC0' trans_port='RE_MantenerCargo' trans_value=''>
                  <trans_to state = 'MC1' />
               </transition>
               <transition trans_from='MC1' trans_port='RE_ActReclamo' trans_value=''>
                  <trans_to state = 'MC2' />
               </transition>
               <transition trans_from='MC2' trans_port='RE_InhibirFactura' trans_value=''>
                  <trans_to state = 'RE2' />
               </transition>
               <transition trans_from='MC3' trans_port='RE_ImprimirCarta' trans_value=''>
                  <trans_to state = 'MC4' />
               </transition>
```

```
        <transition trans_from='AJ0' trans_port='RE_Ajustar' trans_value=''>
            <trans_to state = 'AJ1' />
        </transition>
        <transition trans_from='AJ1' trans_port='RE_Refacturar' trans_value=''>
            <trans_to state = 'AJ2' />
        </transition>
        <transition trans_from='AJ2' trans_port='RE_ActReclamo' trans_value=''>
            <trans_to state = 'AJ3' />
        </transition>
    </behavior>
</component>
<component name='Factura' type='reentrant'>
    <interface>
        <port name='FA_DatosFactura' type='input' min='1' max='1' />
        <port name='FA_LeerFactura' type='output' min='1' max='1' />
        <port name='FA_InhibirFactura' type='input' min='1' max='1' />
        <port name='FA_ActFactura' type='output' min='1' max='1' />
        <port name='FA_ImprimirFactura' type='output' min='1' max='1' />
        <port name='FA_Refacturar' type='input' min='1' max='1' />
    </interface>
    <behavior>
        <state name='DF1' token_ini='0' entry_port='' end_state='FALSE' />
        <state name='DF2' token_ini='0' entry_port='' end_state='TRUE' />
        <state name='IF1' token_ini='0' entry_port='' end_state='FALSE' />
        <state name='IF2' token_ini='0' entry_port='' end_state='TRUE' />
        <state name='DF0' token_ini='0' entry_port='FA_DatosFactura' end_state='FALSE' />
        <state name='IF0' token_ini='0' entry_port='FA_InhibirFactura' end_state='FALSE' />
        <state name='RE0' token_ini='0' entry_port='FA_Refacturar' end_state='FALSE' />
        <state name='RE1' token_ini='0' entry_port='' end_state='FALSE' />
        <state name='RE2' token_ini='0' entry_port='' end_state='FALSE' />
        <state name='RE3' token_ini='0' entry_port='' end_state='TRUE' />
        <transition trans_from='DF1' trans_port='FA_LeerFactura' trans_value=''>
            <trans_to state = 'DF2' />
        </transition>
        <transition trans_from='IF1' trans_port='FA_ActFactura' trans_value=''>
            <trans_to state = 'IF2' />
        </transition>
        <transition trans_from='DF0' trans_port='FA_DatosFactura' trans_value=''>
            <trans_to state = 'DF1' />
        </transition>
        <transition trans_from='IF0' trans_port='FA_InhibirFactura' trans_value=''>
            <trans_to state = 'IF1' />
        </transition>
        <transition trans_from='RE0' trans_port='FA_Refacturar' trans_value=''>
            <trans_to state = 'RE1' />
        </transition>
        <transition trans_from='RE1' trans_port='FA_ActFactura' trans_value=''>
            <trans_to state = 'RE2' />
        </transition>
        <transition trans_from='RE2' trans_port='FA_ImprimirFactura' trans_value=''>
            <trans_to state = 'RE3' />
        </transition>
    </behavior>
</component>
<component name='DB GC' type='repository'>
    <interface>
        <port name='GC_ActReclamo' type='input' min='1' max='1' />
    </interface>
    <behavior>
```

```
            <state name='GC1' token_ini='1' entry_port='' end_state='FALSE' />
            <state name='GC2' token_ini='0' entry_port='' end_state='FALSE' />
            <transition trans_from='GC1' trans_port='GC_ActReclamo' trans_value=''>
               <trans_to state = 'GC2' />
            </transition>
            <transition trans_from='GC2' trans_port='GC_ActReclamo' trans_value=''>
               <trans_to state = 'GC1' />
            </transition>
         </behavior>
      </component>
      <component name='DB FC' type='repository'>
         <interface>
            <port name='FC_LeerFactura' type='read' min='1' max='1' />
            <port name='FC_ActFactura' type='input' min='1' max='1' />
         </interface>
         <behavior>
            <state name='FC1' token_ini='1' entry_port='' end_state='FALSE' />
            <state name='FC2' token_ini='0' entry_port='' end_state='FALSE' />
            <transition trans_from='FC1' trans_port='FC_ActFactura' trans_value=''>
               <trans_to state = 'FC2' />
            </transition>
            <transition trans_from='FC2' trans_port='FC_ActFactura' trans_value=''>
               <trans_to state = 'FC1' />
            </transition>
         </behavior>
      </component>
      <component name='Impresora GC' type='reentrant'>
         <interface>
            <port name='IG_Imprimir' type='input' min='1' max='1' />
         </interface>
         <behavior>
            <state name='IG1' token_ini='0' entry_port='IG_Imprimir' end_state='FALSE' />
            <state name='IG2' token_ini='0' entry_port='' end_state='TRUE' />
            <transition trans_from='IG1' trans_port='IG_Imprimir' trans_value=''>
               <trans_to state = 'IG2' />
            </transition>
         </behavior>
      </component>
      <component name='Impresora FC' type='reentrant'>
         <interface>
            <port name='IF_Imprimir' type='input' min='1' max='1' />
         </interface>
         <behavior>
            <state name='IF1' token_ini='0' entry_port='IF_Imprimir' end_state='FALSE' />
            <state name='IF2' token_ini='0' entry_port='' end_state='TRUE' />
            <transition trans_from='IF1' trans_port='IF_Imprimir' trans_value=''>
               <trans_to state = 'IF2' />
            </transition>
         </behavior>
      </component>
   </components>
   <connectors>
      <connector name='FA_Imprimir_IF' type='queue'/>
      <connector name='RE_Imprimir_IG' type='queue'/>
      <connector name='RE_ActReclamo_GC' type='write'/>
      <connector name='FA_ActFactura_FC' type='write'/>
      <connector name='FA_LeerFactura_FC' type='read'/>
      <connector name='RE_DatosFactura_FA' type='call'/>
      <connector name='RE_Inhibir_FA' type='call'/>
```

```
    <connector name='RE_Refacturar_FA' type='call'/>
</connectors>
<configuration>
    <connection name='FA_Imprimir_IF' ref='FA_Imprimir_IF'>
        <role name='source'>
            <conn_port name='FA_ImprimirFactura'/>
        </role>
        <role name='destination'>
            <conn_port name='IF_Imprimir'/>
        </role>
    </connection>
    <connection name='RE_Imprimir_IG' ref='RE_Imprimir_IG'>
        <role name='source'>
            <conn_port name='RE_ImprimirCarta'/>
        </role>
        <role name='destination'>
            <conn_port name='IG_Imprimir'/>
        </role>
    </connection>
    <connection name='RE_ActReclamo_GC' ref='RE_ActReclamo_GC'>
        <role name='client'>
            <conn_port name='RE_ActReclamo'/>
        </role>
        <role name='repository'>
            <conn_port name='GC_ActReclamo'/>
        </role>
    </connection>
    <connection name='FA_ActFactura_FC' ref='FA_ActFactura_FC'>
        <role name='client'>
            <conn_port name='FA_ActFactura'/>
        </role>
        <role name='repository'>
            <conn_port name='FC_ActFactura'/>
        </role>
    </connection>
    <connection name='FA_LeerFactura_FC' ref='FA_LeerFactura_FC'>
        <role name='client'>
            <conn_port name='FA_LeerFactura'/>
        </role>
        <role name='repository'>
            <conn_port name='FC_LeerFactura'/>
        </role>
    </connection>
    <connection name='RE_DatosFactura_FA' ref='RE_DatosFactura_FA'>
        <role name='source'>
            <conn_port name='RE_DatosFactura'/>
        </role>
        <role name='destination'>
            <conn_port name='FA_DatosFactura'/>
        </role>
    </connection>
    <connection name='RE_Inhibir_FA' ref='RE_Inhibir_FA'>
        <role name='source'>
            <conn_port name='RE_InhibirFactura'/>
        </role>
        <role name='destination'>
            <conn_port name='FA_InhibirFactura'/>
        </role>
    </connection>
```

```
    <connection name='RE_Refacturar_FA' ref='RE_Refacturar_FA'>
      <role name='source'>
        <conn_port name='RE_Refacturar'/>
      </role>
      <role name='destination'>
        <conn_port name='FA_Refacturar'/>
      </role>
    </connection>
  </configuration>
  <ConnectorTypes>
    <ConnType name='Arbitrator' type='built-in' id='arbitrator'>
      <interface>
        <role name='source' min='2' max=''/>
        <role name='destination' min='1' max='1'/>
      </interface>
    </ConnType>
    <ConnType name='Call' type='built-in' id='call'>
      <interface>
        <role name='source' min='1' max='1'/>
        <role name='destination' min='1' max='1'/>
      </interface>
    </ConnType>
    <ConnType name='Call_Any' type='built-in' id='call_any'>
      <interface>
        <role name='source' min='1' max='1'/>
        <role name='destination' min='2' max=''/>
      </interface>
    </ConnType>
    <ConnType name='Datagram' type='built-in' id='datagram'>
      <interface>
        <role name='source' min='1' max='1'/>
        <role name='destination' min='1' max='1'/>
      </interface>
    </ConnType>
    <ConnType name='Fork' type='built-in' id='fork'>
      <interface>
        <role name='source' min='1' max='1'/>
        <role name='destination' min='1' max='1'/>
      </interface>
    </ConnType>
    <ConnType name='Multi_Write' type='built-in' id='multi_write'>
      <interface>
        <role name='client' min='1' max='1'/>
        <role name='repository' min='2' max=''/>
      </interface>
    </ConnType>
    <ConnType name='Publish' type='built-in' id='publish'>
      <interface>
        <role name='publisher' min='1' max='1'/>
      </interface>
    </ConnType>
    <ConnType name='Queue' type='built-in' id='queue'>
      <interface>
        <role name='source' min='1' max='1'/>
        <role name='destination' min='1' max='1'/>
      </interface>
    </ConnType>
    <ConnType name='Queue_All' type='built-in' id='queue_all'>
      <interface>
```

```
            <role name='source' min='1' max='1'/>
            <role name='destination' min='2' max=''/>
         </interface>
      </ConnType>
      <ConnType name='Read' type='built-in' id='read'>
         <interface>
            <role name='client' min='1' max='1'/>
            <role name='repository' min='1' max='1'/>
         </interface>
      </ConnType>
      <ConnType name='Rendez-Vous' type='built-in' id='rendez-vous'>
         <interface>
            <role name='source' min='1' max='1'/>
            <role name='destination' min='1' max='1'/>
         </interface>
      </ConnType>
      <ConnType name='Subscribe' type='built-in' id='subscribe'>
         <interface>
            <role name='subscriber' min='1' max='1'/>
         </interface>
      </ConnType>
      <ConnType name='Write' type='built-in' id='write'>
         <interface>
            <role name='client' min='1' max='1'/>
            <role name='repository' min='1' max='1'/>
         </interface>
      </ConnType>
   </ConnectorTypes>
</architecture>
</SolutionXML>
```

Tesis de Licenciatura

Definición e Implementación de un Lenguaje de Descripción de Arquitecturas de Software

Anexo II – Evolución del trabajo

Se intentan destacar en este Anexo -divididos por "secuencias de discusiones"- los cambios más relevantes que condujeron a la versión final de la Tesis aquí presentada, mostrados a modo de enunciados, partiendo de la versión inicial de JACAL y de nuestro Proyecto de Tesis presentado oportunamente.

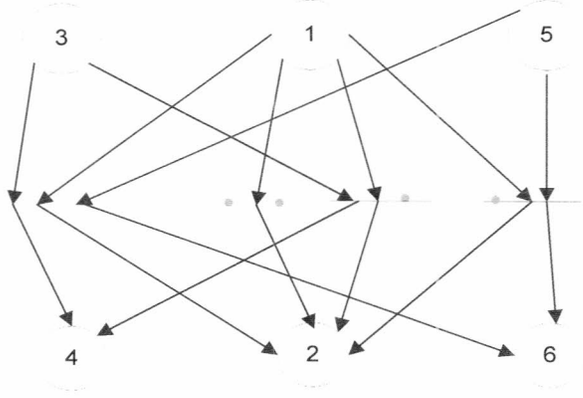
Todos los pasos en esta Evolución (o sea, cada una de estas "discusiones") tuvieron lugar dentro del Grupo de Tesis, con consultas a (y/o bajo la guía de) nuestro Director de Tesis, Lic. Nicolás Kicillof, e inicialmente también con el Dr. Víctor Braberman; y muchas de estas discusiones conceptuales se hicieron en base a representaciones presentadas gráficamente en Drawings de MS Visio con formatos similares al siguiente gráfico, mostrado como ejemplo:

Para **Datagram**:

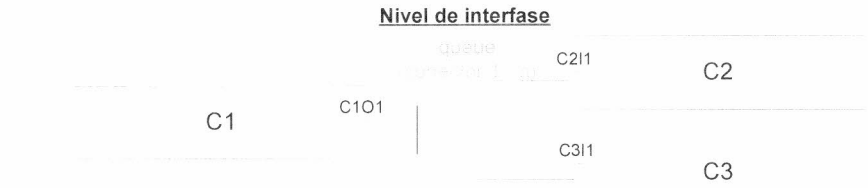
Dinámica pretendida:
1) C1 envía mensaje por *datagram* a C2 y C3.
2) Simultáneamente:
   *) C2 estaba esperando el mensaje, y lo recibe;
   *) C3 no estaba esperando el mensaje, y lo pierde;
   *) C1 continúa ejecutando.

Nivel de comportamiento

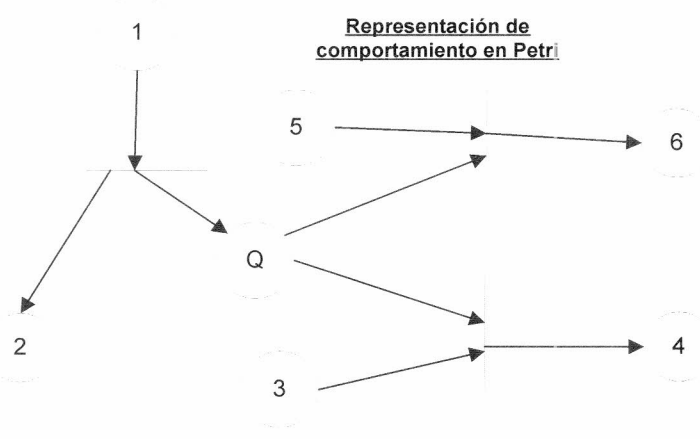
Representación de comportamiento en Petri



Para **Queue**:



Dinámica pretendida:
1) C1 envía mensaje por *queue* a C2 y C3.
2) Simultáneamente:
*) El mensaje es dejado en la cola central;
*) C1 continúa ejecutando.
[3) En el momento en que algún receptor -aquí se ejemplifica con C2- consulte la cola:
a) si el mensaje estaba y él es el único que lo estaba esperando, lo consumirá; o
b) si el mensaje estaba y hay más componentes esperándolo, se decidirá no determinísticamente quién lo consumirá (los no elegidos seguirán bloqueados); o bien
c) si no había ningún mensaje, se queda bloqueado hasta que lo haya.
(También se muestra al receptor C3 que no consulta su cola y continúa ejecutando).
4) Luego de consumir el mensaje, C2 continúa ejecutando]



y así con el resto de los conectores.

- Versión básica inicial XML, heredada de la versión anterior de JACAL.
- Análisis de alternativas de notación para representación de Comportamiento: Redes de Petri, o Autómatas Temporizados (formalismo de base para varios trabajos previos del grupo de investigación, que no tiene una notación textual

estándar, están muy difundidos y hay varias herramientas que los aceptan como input, incluyendo herramientas de model checking). // Idea: que los conectores soporten un estado "desconectado" con una semántica diferente de la habitual y que serviría para simular redes móviles, donde la conexión no es permanente. [En la aplicación, se podría "desconectar" en forma manual (desconectando durante la ejecución). Pero también se podría hacer que la desconexión responda a algo que pase en el sistema, así se pueden desconectar varios conectores a la vez sin tener que hacerlo uno por uno y también en respuesta a determinados eventos.]

- Incorporación de la noción de roles para ports en los conectores (ejs: source, destination) // Necesidad de representar el comportamiento de los componentes –idea del tag <macro> en XML- con alguna notación estándar. // Idea de crear tipos de conectores custom (al que se le pudiera definir el comportamiento de manera similar al de los componentes, para lograr extensibilidad para el lenguaje) y lock, que bloquearía al componente destino en caso de recibirse determinado flujo de ejecución por este medio, y que le indicaría que ya está siendo utilizado por otro proceso. // Búsqueda de coherencia conceptual entre Tipo de Componente y Tipo de Conector: incluiría su nomenclatura, aridad y aspectos de su comportamiento. // Idea para clasificación de Tipos de Componentes (asociados a su semántica más que a su interfase): statefull vs. stateless + activos vs. pasivos; el tipo de componente daría de alguna manera la semántica que tiene su comportamiento. Por ejemplo, qué hace cuando recibe un flujo de ejecución y ya tiene otro adentro. // Concepto de evento absorbido por conector publish/subscribe. // Se descarta trabajar con nociones temporales. // Se asume un único broker, que cumpliría la función de un bus de información implícito en el sistema. // Idea de port de I/O, y de distinción entre flujo de datos y de control a nivel de roles. // ¿Nos sirve un conector callback (símil publish/subscribe sincrónico)? // Análisis de conectores "árbitros" con representación de sincronización. // ¿Incorporamos un estado (activo/inactivo) a los conectores? // Reevaluación de conector interrupt (sería innecesario, ya que el Tipo del componente receptor podría determinar la interrupción de un flujo de ejecución). // Noción de Configuración, independiente de Conectores y Componentes, en la que se definirían cosas tales como qué port de qué componente cumple qué rol de qué conector.
- Factibilidad de conectores *write* y *read* asincrónicos, donde en ambos casos no se esperaría el resultado de la operación sino que se continuarían ejecutando

otros procesos // Factibilidad de implementación de *callback*, con el cual el componente emisor podría informar a un componente receptor que conoce la dirección o punto de entrada de una función del emisor, (sin conoce al emisor mismo) para que el receptor lo llame ante algún evento; o su uso conceptual como herramienta para la implementación del conector *subscribe*.

- Reemplazo de las transiciones etiquetadas con un port de return (prima) en JACAL 1 por un atributo adicional en el port (*return="yes"*) para que el *return* no dependa de cuál fue el port desde el cual se recibió el *call*, que complicaría en gran modo las validaciones; o bien su eliminación, por lo menos para los casos en que no hay que devolver datos.
- Revisión de representaciones de funcionalidad de conectores con Redes de Petri debido a su excesiva simplificación inicial: en el *queue*, las transiciones deberían llegar directamente a los nodos que representan las colas, no a los estados del siguiente componente; en el *fork*, se debería ver más claramente cómo se eligen los nodos de destino; en el *call*, no se veía claramente la secuencia del comportamiento en el componente destino así como la posible devolución de un valor que afecte el comportamiento del emisor; en el *syncDataAccess* (*read* + *write*) ídem último problema mencionado para el *call*, además de que no se distinguían las funcionalidades intrínsecas del *read* de las del *write*; el *publish/subscribe* debería manejar el concepto de evento de modo similar a como se hacía en JACAL 1: si se produce el evento, se debería poner 1 token en el "depósito" de eventos (similar a las queues), de modo que si hay alguien suscripto (lo cual también tiene que indicarse con 1 token en un depósito de suscripciones) reciba el mensaje.
- Edición de arquitecturas y simulación: decisión de utilizar MS Visio 2002, con soporte del SKD de Visio, y en código VBA bajo el modelo MVC (Model View Controller).
- Confirmación de que la representación en Petri no modelará "esperas" (estados intermedios para mostrar pasos en una secuencia) en comportamientos asincrónicos // *datagram* tendría 2 roles: emisor y receptor(es); y la conexión entre ellos sería estática, determinada por la configuración de la arquitectura, habría un port cumpliendo cada uno de los roles // En el *publish/subscribe*, el suscriptor a un evento indicaría de alguna forma su interés de suscribirse en forma dinámica, proceso que podría depender del nombre del "tema" al que se suscribe (no del nombre del port); o sea que un suscriptor en determinado momento pediría que determinado port suyo se suscriba a determinado tema

(por nombre), resultando en una transición que tendría como etiqueta "subscribe port <p> to subject <s>".

- Distinción conceptual entre componentes *stateful* y "demonios" o servicios (que están siempre activos) // Innecesidad de distinción entre almacenamiento de datos y de control dentro de un componente // No incluir en este trabajo a los conectores *custom*, entendiendo por éstos a aquellos conectores cuyo comportamiento se especificará a través de algún archivo externo, o dentro del mismo esquema pero como alguna combinación de otros elementos ya existentes. // Se pensarán todos los flujos como flujos de control (o *flujos de ejecución*), terminando con la ambigüedad de JACAL 1, que los mezclaba con los flujos de datos. // Descarte de conector *kill*
- Introducción de conector *callAny*, como un *call* que pueda responder cualquiera de los invocados, cuyo comportamiento podría describirse con el no determinismo de las Redes de Petri (contesta cualquiera, sin arcos negativos).
- *publish/subscribe* en Redes de Petri: si el *componente* C puede querer suscribirse al tema T, deberíamos tener un place que sea <C,T> y poner un token ahí cuando se suscribe; si se desuscribe, sacar el token; si hay un *publish* dirigido a un tema, deberíamos mandarle la publicación al *componente* C si y sólo si hay un token en <C,T>. Y los ports que se usan en los *componentes* suscriptores tienen que ser de entrada Y salida (I/O). // Necesidad de incluir *transiciones fork* (transiciones que hicieran *fork* dentro del componente, quedando visualmente como una Y, y que duplicarían tokens, de modo de poder representar situaciones como la de un componente que viene ejecutando, decida ponerse a escuchar en un port –ej: de una cola o de un *datagram*, y además siga su ejecución) y componentes que empiecen con un flujo de ejecución sin que haga falta inicializarlos.
- Interfase de *subscribe*: el *conector* debería llegar a UN SOLO port (que será de I/O, porque por ahí se envía el mensaje de suscripción y se reciben las publicaciones) en el *componente* que se suscribe, con lo cual surge la idea de ponerle un signo de exclamación (ej: C2!) a las transiciones que envían un mensaje por un port y un signo de interrogación (ej: C2?) a las que reciben.
- Corrección de nomenclatura de ports: C2I1 para "port de input 1 del componente 2", para que el nombre del port no lleve el nombre del conector o del tema, lo que provocaría un fuerte acoplamiento; y manejando los signos de exclamación e interrogación como se explicó antes.

- Introducción de *entry point* como un nuevo tipo de port que reemplace la funcionalidad del *estado inicial* en JACAL 1, con similares restricciones de comportamiento (que la transición que tiene como label a este port –que estaría conectado a un *call*- no pueda recibir un flujo de ejecución por otro lado, o sea que tendría que salir de un estado al que no llegaran otras transiciones), y que además desde el punto de vista de la interfase sólo pueda tener el rol de destino en determinados conectores (*call*, *fork*, *callAny*).
- Idea de *queueAny,* de cardinalidad 1-n, con el que se podría dejar un mensaje en una cola general (y si un destinatario consume el mensaje, el resto no). // Reconsideración del tipo de port que cumple el rol de suscriptor en un *publish/subscribe*: en lugar de I/O, tipo especial *subscribe(S)* porque funciona como *Output* para la suscripción, pero como *entry* para la recepción de publicaciones. // Idea para el caso de *entry*: en lugar de que haya un nodo inicial, que sea directamente una transición que no tenga origen.
- Reconsideración de clasificación de ports por sus restricciones: sobre el nivel de comportamiento del *componente*, sobre los posibles *conectores* y sobre el rol que puede cumplir en un conector. // Mantener el criterio de comportamiento no determinístico en *componentes* de JACAL 1.
- Idea de permitir return en el *fork*, o ports *entry* en *conectores* como *queue* o *datagram*, de modo de seguir facilitando las combinaciones posibles entre *conectores* y *componentes*. // Idea de extensión de Redes de Petri también para tipos de ports, para que éstos hagan parte de la tarea de los *conectores* (ej: poder hacer que desde el punto de vista del *conector* los ports de input fueran todos iguales, pero cada port se encargaría de hacer su trabajo de acuerdo al comportamiento que se les va a asociar en el *componente*). // ¿Redenominación de *multiWrite* por *writeAll*? ¿O reemplazar el *multiWrite* directamente por un *callAll*, que llamaría a varios *componentes* y la ejecución del emisor no seguiría hasta que no hayan contestado todos? (En este caso, el valor de retorno podría ser el del último, y el *callAll* se podría usar para cualquier tipo de componente/port, descartando la posible respuesta).
- Énfasis en tratar de utilizar formas estándar para los *conectores*. // Permitir que haya varios *conectores* que lleguen a 1 port de input, para reflejar casos como el de un sistema que llame a la misma función o método desde distintos lugares, lo cual se traduciría en varios *calls* que llegan a un mismo port de input. // Idea de un nuevo *conector* (*enable*) que suscriba a otro a un tema dado, para reflejar casos en los que un *componente* no sabe que va a recibir las llamadas

mediante una cola, con lo cual no puede suscribirse él mismo: tiene que suscribirlo otro. // Idea de cambiar *publish/subscribe* por sólo *publish* (que seguiría permitiendo que el receptor se des/suscriba al tema), con el objeto de agregar también otro *conector subscribe* que llegue al mismo port de input que el *publish* pero que permita des/suscribir al receptor a ese tema.

- Notación: place de un port como subíndice de su nombre, y valores de retorno como supraíndices.
- Readaptación de las Redes de Petri diseñadas a formatos similares a Redes de Petri coloreadas (que incluyan o reflejen "tipación"), de modo que los colores puedan asociarse a Tipos de Datos que tendrían información implícita tal como el port de input o el conector al que responder en un *call*, e incluso permitan sentencias condicionales, asimismo como una representación de pila para poder seguir la secuencialidad de los tokens en aquellos casos en los que varios conectores se conectan al mismo port de input y respetar esa secuencia para la respuesta ordenada. // Eliminación de toda consideración o referencia a que algo se resolverá "por programa", debiendo representarse esta operatoria en la semántica. // Tratamiento de *return* por *estado terminal* (manejo de excepciones): uso de pattern matching en notación para garantizar la correspondencia del *return* con la transición del port.
- Nuevo *conector* (*rendez-vous*) que permite sincronizar el flujo de ejecución de 2 componentes.
- Forma tentativa de que un *componente* se suscriba a un tema: recibiendo un *subscribe* (en vez de *enable*) por el mismo port por el que se reciben las publicaciones. Y si se quisiera suscribir solo (sin ayuda de otro), él mismo –desde un port de output propio- emitiría el *subscribe* a su port de input.
- Se eliminan los *componentes no reentrantes* a efectos de simplificar el lenguaje.
- Incorporación de *componentes repository*, cuyo comportamiento permitiría hacer una 'traducción' con transiciones que cambien el valor de los tokens para el *write*, por ejemplo.
- Introducción de tipo de port *read*, asociable sólo a los *componentes repository*, y permitiendo sólo 1 de estos ports por componente // Semántica de port *read*: se asigna el nombre del estado (valor del token) transferido al port, ubicación sobre la cual recupera la información el *conector read* asociado (único tipo de *conector* asociable a este tipo de port).

Tesis de Licenciatura

Definición e Implementación de un Lenguaje de Descripción de Arquitecturas de Software

Anexo III – Semántica de Ports y Conectores

Se muestran en este Anexo las representaciones de la semántica completa utilizada para reflejar el comportamiento de los Ports y de los Conectores de JACAL 2.0.

Con Drawings de MS Visio como herramienta, fuimos detallando los Tipos de Ports que definimos, las relaciones permitidas entre los Tipos de Ports y los Tipos de Componentes, los distintos *places* ad hoc que definimos para cada Tipo de Port, la representación (en notación Petri) asociada al comportamiento de cada Tipo de Port en cada Tipo de Componente permitido, y finalmente la representación (en notación Petri) del comportamiento de cada Tipo de Conector que definimos para JACAL 2.0:

Tipos de port

El tipo de port asociable a un componente depende del tipo del mismo:
Componentes reentrantes:
*) ports de tipo *input;* y
*) ports de tipo *output*.
Componentes de tipo repositorio:
*) ports de tipo *read*, y
*) ports de tipo *input*.

Para determinar sus comportamientos ante cada posible transición, se definen determinados *places* para cada uno de ellos, a saber:

Ports de tipo input

W → Almacenamiento persistente de tokens en espera

I → Destino de tokens entrantes

R → Retorno de tokens

Ports de tipo output

C → Recepción de continuación

O → Destino de tokens salientes

Ports de tipo read

I → Destino de tokens entrantes

R → Retorno de tokens

TIPOS DE PORTS

Ports de input

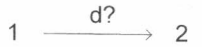
Especificación del comportamiento asociado a las transiciones

Se asume que todos los lugares de las redes de Petri definidas tienen asociado el tipo de dato Flow:
(List<Port> x List<Connector> x Data)

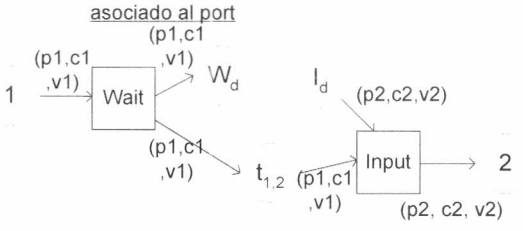
Nivel de comportamiento

Comportamiento asociado al port

I)



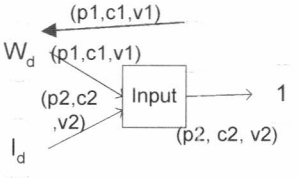
(transición asociada al port d)



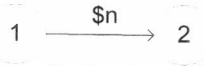
Componente reentrante

II)

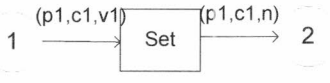
(punto de entrada asociado al port d)



III)

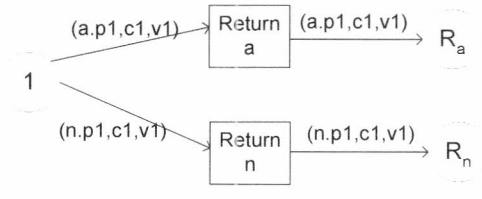


(Asignación de valor al token.
El estado 2 puede o no ser terminal)



IV)

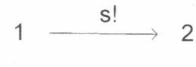
(estado terminal)



Ports de output

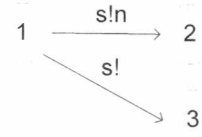
Especificación del comportamiento asociado a las transiciones

Nivel de comportamiento

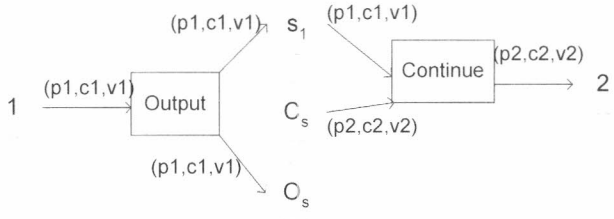


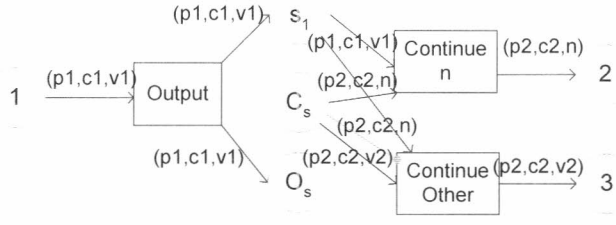
(Transición asociada al port s)

(Transición condicional asociada al port s - Se asume que s! opera como condición "other")



Comportamiento asociado al port





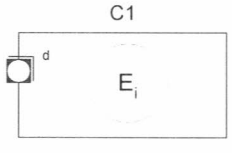
Ports de tipo read

Especificación del comportamiento asociado a las transiciones

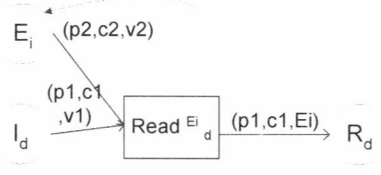
Se asume que todos los lugares de las redes de Petri definidas tienen asociado el tipo de dato Flow:
(List<Port> x List<Connector> x Data)

Los ports de tipo read son asociables solo a componentes de tipo repository, y no se permite instanciar mas de uno por componente. Por cada estado definido en el comportamiento de los componentes de tipo repository, se asumen como definidas en forma implìcita transiciones reflexivas asociadas al ùnico port de tipo read (en caso que exista). Dichas transiciones permitiràn recuperar el estado en el cual se encuentra el componente cada vez que se reciba una solicitud por el port "read".

Nivel de comportamiento



Comportamiento asociado al port

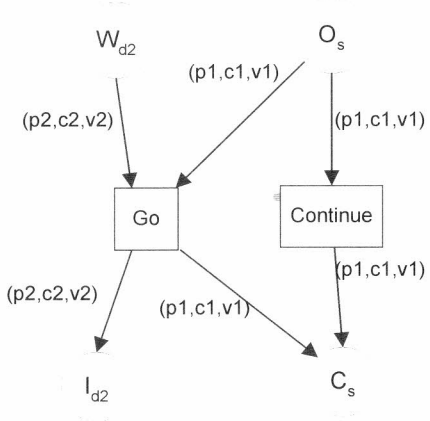


Se asume la existencia de una transición reflexiva implícita, asociada a cada estado del comportamiento de un componente de tipo "repository", por cada port de tipo read.

Nivel de interfase

Dinámica pretendida:
1) C1 envía mensaje por *datagram* a C2.
2) Simultáneamente:
   *) C2 recibe el mensaje, en caso que se encontrara esperándolo.
   *) C2 pierde el mensaje, en caso que no se encontrara esperándolo.
   *) C1 continúa ejecutando siempre.

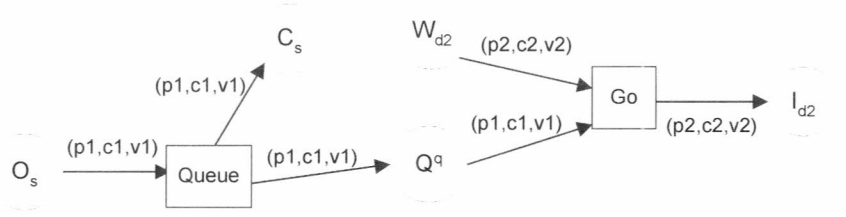
Representación de comportamiento en Petri



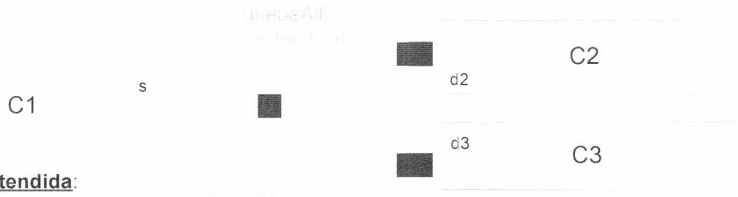
Nivel de interfase

Dinámica pretendida:
1) C1 envía mensaje por *queue* a C2.
2) Simultáneamente:
*) El mensaje es dejado en la cola central;
*) C1 continúa ejecutando.
[3) En el momento en que algún receptor -aquí se ejemplifica con C2- consulte la cola:
a) si el mensaje estaba y él es el único que lo estaba esperando, lo consumirá; o
b) si el mensaje estaba y hay más componentes esperándolo, se decidirá no determinísticamente quién lo consumirá (los no elegidos seguirán bloqueados); o bien
c) si no había ningún mensaje, se queda bloqueado hasta que lo haya.
4) Luego de consumir el mensaje, C2 continúa ejecutando]

Representación de comportamiento en Petri



Nivel de interfase



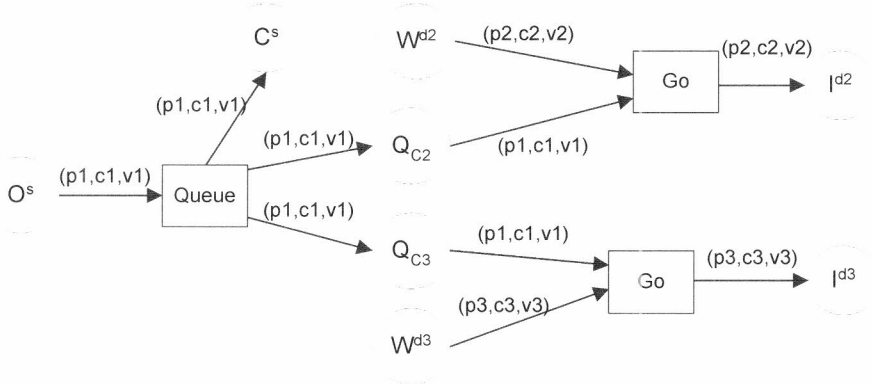
Dinámica pretendida:

1) C1 envía mensaje por *queueAll* a C2 y C3.
2) Simultáneamente:
   *) C2 y C3 reciben los mensajes en sus respectivas colas;
   *) C1 continúa ejecutando.

[3) En el momento en que algún receptor -aquí se ejemplifica con C2- consulte su cola:
   a) si el mensaje estaba, lo consumirá; o bien
   b) si por alguna razón no había ningún mensaje, se queda bloqueado hasta que lo haya.
(También se muestra al receptor C3 que no consulta su cola y continúa ejecutando).
4) Luego de consumir el mensaje, C2 continúa ejecutando]

Nivel de comportamiento

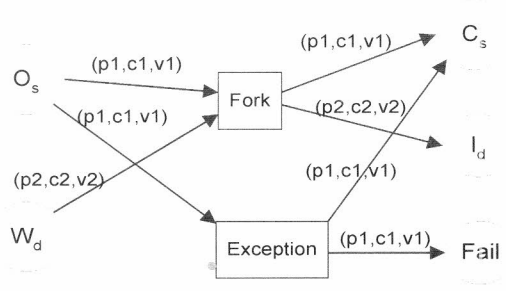
Representación de comportamiento en Petri



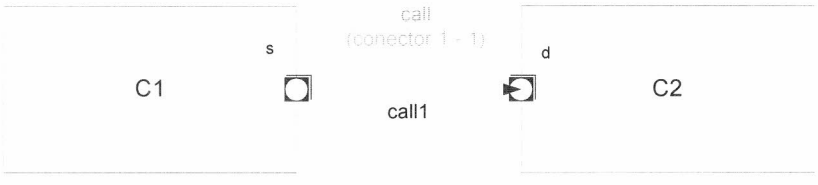
Nivel de interfase

Dinámica pretendida:
1) C1 envía mensaje por *fork* a C2, C3, etc.
2) Simultáneamente:
*) C2, C3, etc. empiezan a ejecutar;
*) C1 continúa ejecutando.

Representación de comportamiento en Petri

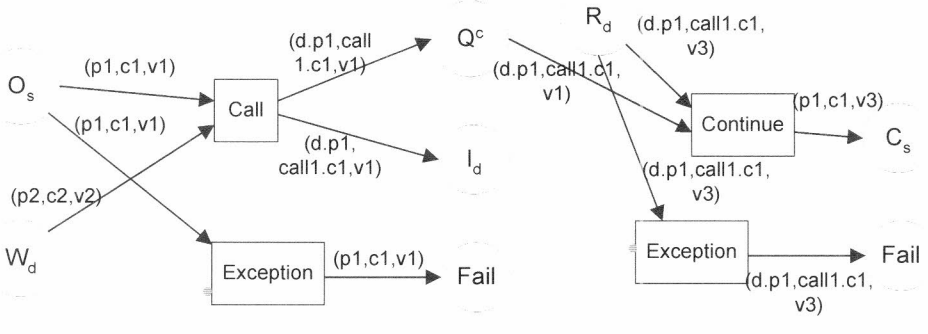


Nivel de interfase



Dinámica pretendida:
1) C1 envía mensaje por *call* a C2.
2) C2 empieza a ejecutar.
3) C2 devuelve resultado a C1, que continúa ejecutando de acuerdo con el valor devuelto.

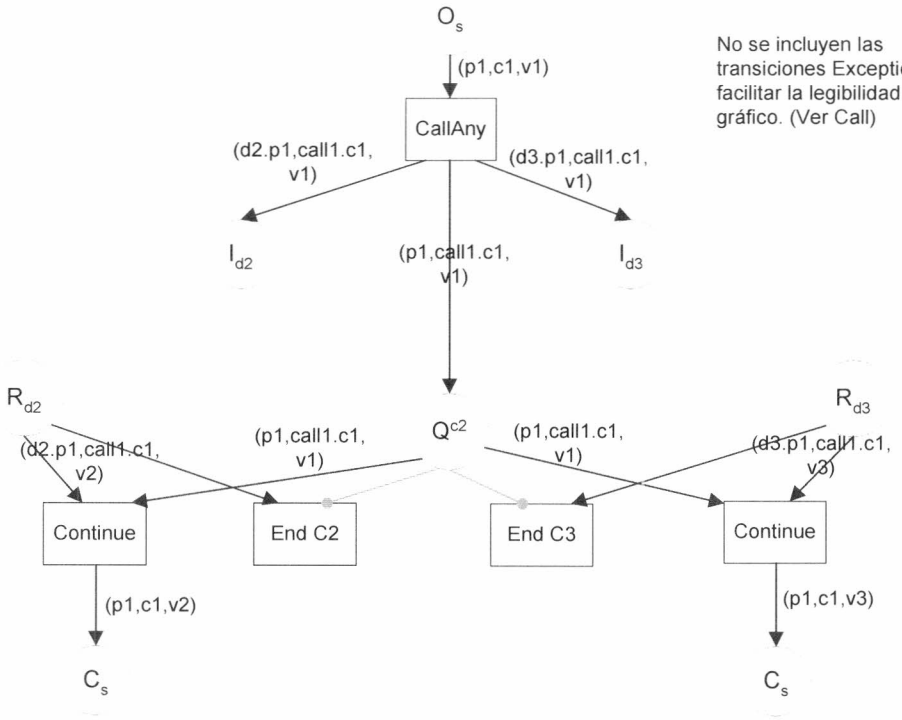
Representación de comportamiento en Petri



Nivel de interfase

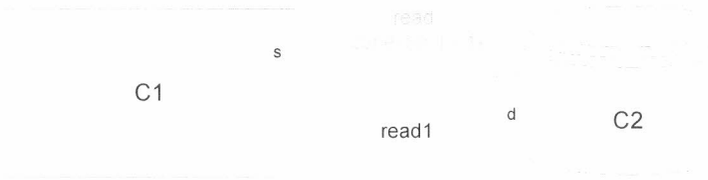
Dinámica pretendida:
(La idea general es que se comporte como una combinación de un *cali* múltiple con un *publish* sincrónico hasta que responda 1 receptor)
1) C1 envía mensaje por *callAny* al Administrador ( ).
2) El Administrador llama a los destinatarios pretendidos C2, C3, ... etc.
3) C2 es el primer destinatario que devuelve resultado al Administrador, y continúa ejecutando.
4) El Administrador le devuelve el resultado a C1.
5) C1 continúa ejecutando.

Representación de comportamiento en Petri



No se incluyen las transiciones Exception para facilitar la legibilidad del gráfico. (Ver Call)

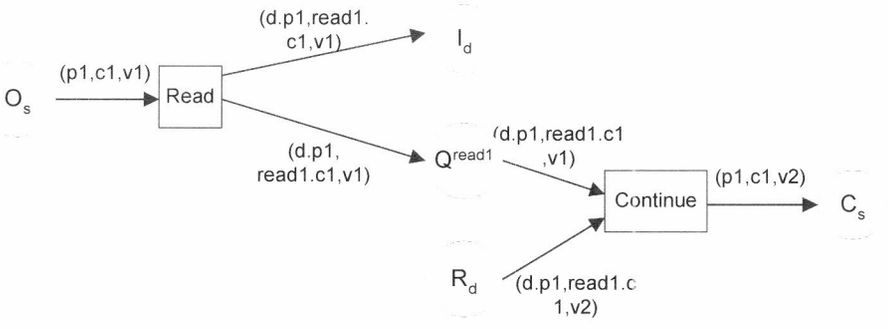
Nivel de interfase



Dinámica pretendida:
1) C1 envía mensaje por *read* a C2.
2) C2 devuelve su estado actual a C1, que continúa ejecutando de acuerdo con el resultado devuelto. C2 debe ser un componente de tipo repositorio, el conector debe asociarse al port de tipo "read" de C2, y el valor del token corresponde al nombre del estado interno de C" en el cual se encuentra en ese momento.

Representación de comportamiento en Petri

Nota: El rol destino de este conector solo puede asociarse a un componente de tipo "repository".

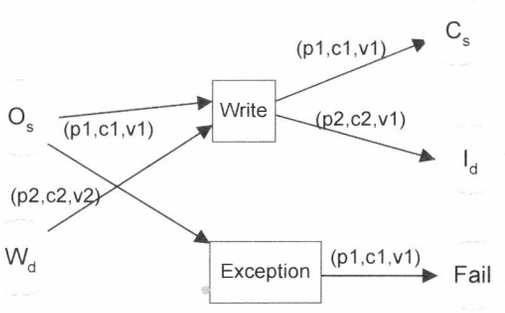


Nivel de interfase

Dinámica pretendida:
1) C1 envía mensaje por *write* a C2.
2) C2 cambia su estado, y el resultante lo devuelve a C1, que continúa ejecutando.

Nivel de comportamiento

Representación de comportamiento en Petri



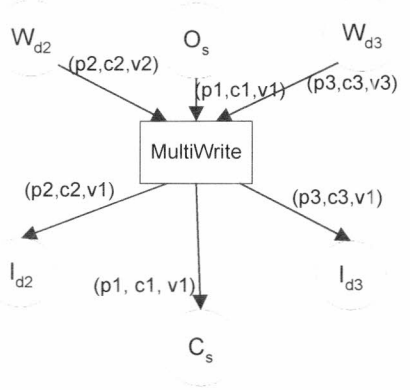
Nota: El rol destino de este componente solo puede asociarse a un componente de tipo "repository".

Nivel de interfase

Dinámica pretendida:
1) C1 envía mensaje por *multiWrite* a C2, C3, ... etc.
2) El *multiWrite* determina que se le asignen **TODOS** los recursos (C2, C3,... etc.) a C1.
3) C1 utiliza C2, C3, ... etc.
4) C1 continúa ejecutando.

Nivel de comportamiento

No se incluyen las transiciones Exception para facilitar la legibilidad del gráfico. (Ver Write)



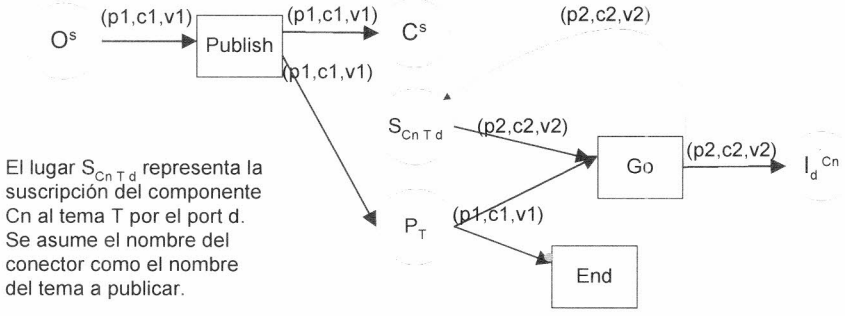
Nota: El rol destino de este componente solo puede asociarse a componentes de tipo "repository".

Nivel de interfase

Dinámica pretendida para la función publish:
1) C1 (publicador) envía mensaje por *publish* sobre el tema T y continúa ejecutando.
2) El Administrador (*no representado*) envía el mensaje a C2, C3, ... etc (que estaban suscriptos al tema T) por sus ports d2, d3, ... etc, que les queda disponible mientras sigan suscriptos a T.

Nivel de comportamiento - función *publish*

Representación de comportamiento en Petri



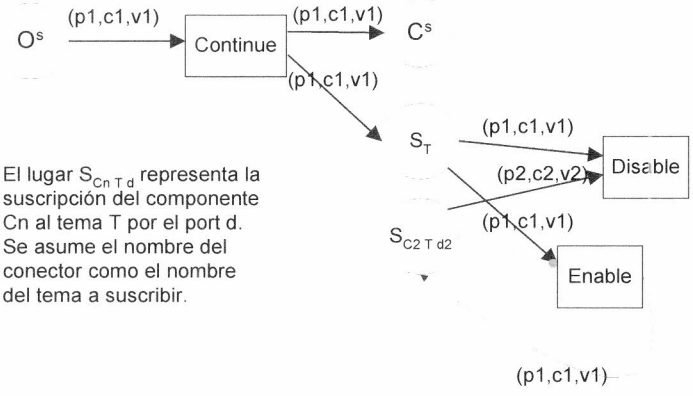
El lugar $S_{Cn\ T\ d}$ representa la suscripción del componente Cn al tema T por el port d. Se asume el nombre del conector como el nombre del tema a publicar.

Nivel de interfase

Dinámica pretendida para la función *(un)subscribe*:
1) C1 envía mensaje por *subscribe* al tema T para (de-)suscribir al port destino (que puede corresponder al mismo componente o bien a un tercero) y continúa ejecutando.
2) El Administrador (no representado) actualiza su información interna (publicadores - temas - suscriptores).

Nivel de comportamiento - función *enable*

Representación de comportamiento en Petri

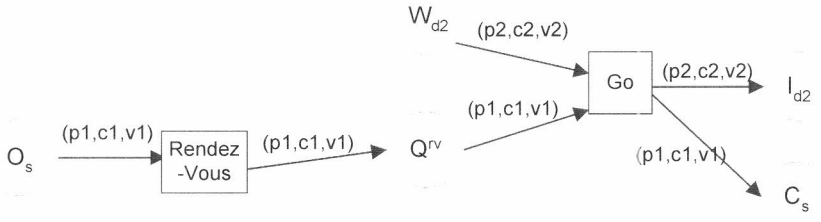


El lugar $S_{Cn\ T\ d}$ representa la suscripción del componente Cn al tema T por el port d. Se asume el nombre del conector como el nombre del tema a suscribir.

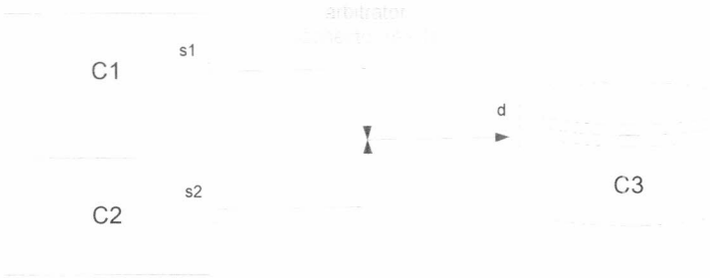
Nivel de interfase

Dinámica pretendida:
1) C1 envía mensaje por *rendez-vous* a C2.
2) C1 queda bloqueado, hasta tanto C2 no se encuentre disponible para ejecutar.
3) Cuando C2 se encuentra disponible, C1 y C2 sincronizan su ejecución, continuando simultáneamente.

Representación de comportamiento en Petri

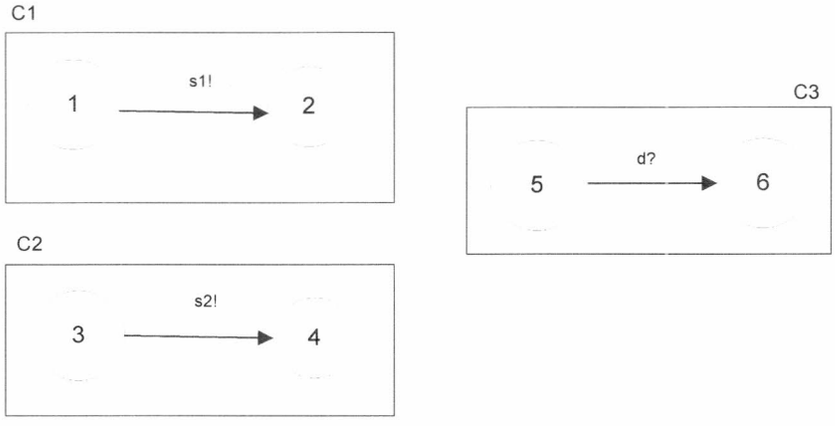


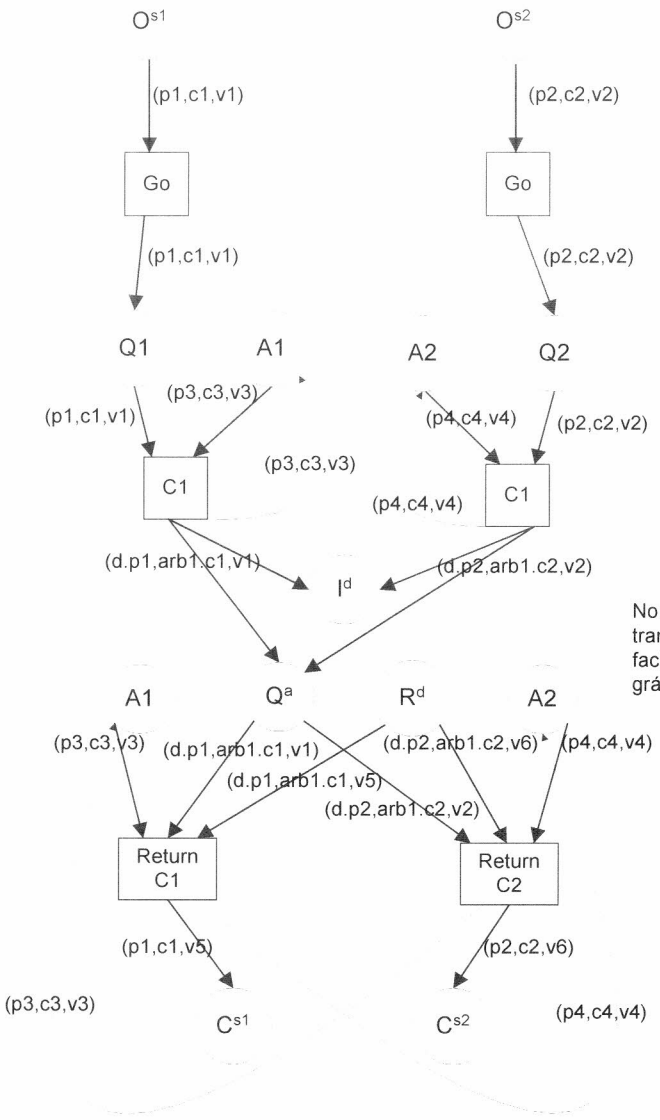
Nivel de interfase



Dinámica pretendida (con 2 emisores):
1) C1, C2,... envían mensaje por *arbitrator* a C3.
2) El *arbitrator* determina que se le asigne el recurso a C1.
3) Si C1 tenía algún pedido encolado en ese momento, lo utiliza, y continúa ejecutando.
4) El *arbitrator* determina que se le asigne el recurso a C2.
5) Si C2 tenía algún pedido encolado en ese momento, lo utiliza, y continúa ejecutando.
etc.

Nivel de comportamiento





No se incluyen las transiciones Exception para facilitar la legibilidad del gráfico. (Ver Call)

Tesis de Licenciatura

Definición e Implementación de un Lenguaje de Descripción de Arquitecturas de Software

Anexo IV – Sintaxis JACAL 2.0 en BNF

A continuación, se detalla la especificación genérica del lenguaje JACAL 2.0, en una sintaxis basada en notación BNF (Backus Naus Form):

```
ARCHITECTURE := {<COMPONENTS>, <CONNECTORS>, <CONFIGURATION>,
    <CONNECTOR_TYPES>}

COMPONENTS := {<COMPN>, <COMP_TYPE>}
COMPN := <STRING>
COMP_TYPE := (<comp_t>, <INTERFACE>, <BEHAVIOR>)
comp_t in (reentrant | repository)
INTERFACE := {<PORTN>, <PORT_TYPE>}
PORTN := <STRING>
PORT_TYPE := (<port_t>, <MIN>, <MAX>)
port_t in (INPUT | OUTPUT)
MIN := <NUM>*
MAX := <NUM>*
BEHAVIOR := ({<STATE>},{<TRANSITION>})
STATE := (<STATN>, <TOKEN_INI>, <ENTRY_PORT>, <END_STATE?>)
STATN := <STRING>
TOKEN_INI := <NUM>*
ENTRY_PORT := <STRING>
END_STATE? in (TRUE | FALSE)
TRANSITION := (<STATEFROM>, <PORTN>, <VALUE_TRANS>,
    <STATETO>)
STATEFROM := <STRING>
STATETO := <STRING>
VALUE_TRANS := <NUM>*

CONNECTORS := {<CONNECTORN>, <CONNECTOR_TYPE>}
CONNECTORN := <STRING>
```

```
CONNECTOR_TYPE in (arbitrator | call | call_any | datagram |
    fork | multi_write | publish | queue | queue_all | read |
    rendez-vous | subscribe | write)

CONFIGURATION := {<CONNECTIONNAME>, <CONNECTIONREF>,
    {<CONNECTION>}}
CONNECTIONNAME := <STRING>
CONNECTIONREF := <STRING>
CONNECTION := {<ROLE>}
ROLE := (<ROLENAME>, <CONNECTION_PORT_NAME>)
ROLENAME := <STRING>
CONNECTION_PORT_NAME := <STRING>

CONNECTOR_TYPES := (<BUILT-IN_CONN> | <USER-DEF_CONN>)

BUILT-IN_CONN := (<ARBITRATOR> | <CALL> | <CALL_ANY> |
    <DATAGRAM> | <FORK> | <MULTI_WRITE> | <PUBLISH> | <QUEUE>
    | <QUEUE_ALL> | <READ> | <RENDEZ-VOUS> | <SUBSCRIBE> |
    <WRITE>)

ARBITRATOR := (<ARB_NAME>, <ARB_TYPE>, <ARB_ID>, <ARB_INT>)
ARB_NAME := arbitrator
ARB_TYPE := built-in
ARB_ID := arbitrator
ARB_INT := (<ARB_MIN_SOURCE>, <ARB_MAX_SOURCE>,
    <ARB_MIN_DEST>, <ARB_MAX_DEST>)
ARB_MIN_SOURCE := 2
ARB_MAX_SOURCE := 6
ARB_MIN_DEST := 1
ARB_MAX_DEST := 1

CALL := (<CALL_NAME>, <CALL_TYPE>, <CALL_ID>, <CALL_INT>)
```

```
CALL_NAME := call
CALL_TYPE := built-in
CALL_ID := call
CALL_INT := (<CALL_MIN_SOURCE>, <CALL_MAX_SOURCE>,
    <CALL_MIN_DEST>, <CALL_MAX_DEST>)
CALL_MIN_SOURCE := 1
CALL_MAX_SOURCE := 1
CALL_MIN_DEST := 1
CALL_MAX_DEST := 1

CALL_ANY := (<CALL_ANY_NAME>, <CALL_ANY_TYPE>, <CALL_ANY_ID>,
    <CALL_ANY_INT>)
CALL_ANY_NAME := call_any
CALL_ANY_TYPE := built-in
CALL_ANY_ID := call_any
CALL_ANY_INT := (<CALL_ANY_MIN_SOURCE>,
    <CALL_ANY_MAX_SOURCE>, <CALL_ANY_MIN_DEST>,
    <CALL_ANY_MAX_DEST>)
CALL_ANY_MIN_SOURCE := 1
CALL_ANY_MAX_SOURCE := 1
CALL_ANY_MIN_DEST := 2
CALL_ANY_MAX_DEST := 6

DATAGRAM := (<DAT_NAME>, <DAT_TYPE>, <DAT_ID>, <DAT_INT>)
DAT_NAME := datagram
DAT_TYPE := built-in
DAT_ID := datagram
DAT_INT := (<DAT_MIN_SOURCE>, <DAT_MAX_SOURCE>,
    <DAT_MIN_DEST>, <DAT_MAX_DEST>)
DAT_MIN_SOURCE := 1
DAT_MAX_SOURCE := 1
DAT_MIN_DEST := 1
```

```
DAT_MAX_DEST := 1

FORK := (<FORK_NAME>, <FORK_TYPE>, <FORK_ID>, <FORK_INT>)
FORK_NAME := fork
FORK_TYPE := built-in
FORK_ID := fork
FORK_INT := (<FORK_MIN_SOURCE>, <FORK_MAX_SOURCE>,
    <FORK_MIN_DEST>, <FORK_MAX_DEST>)
FORK_MIN_SOURCE := 1
FORK_MAX_SOURCE := 1
FORK_MIN_DEST := 1
FORK_MAX_DEST := 1

MULTI_WRITE := (<MULTI_W_NAME>, <MULTI_W_TYPE>, <MULTI_W_ID>,
    <MULTI_W_INT>)
MULTI_W_NAME := multi_write
MULTI_W_TYPE := built-in
MULTI_W_ID := multi_write
MULTI_W_INT := (<MULTI_W_MIN_CLIENT>, <MULTI_W_MAX_CLIENT>,
    <MULTI_W_MIN_REPOSITORY>, <MULTI_W_MAX_REPOSITORY>)
MULTI_W_MIN_CLIENT := 1
MULTI_W_MAX_CLIENT := 1
MULTI_W_MIN_REPOSITORY := 2
MULTI_W_MAX_REPOSITORY := 6

PUBLISH := (<PUB_NAME>, <PUB_TYPE>, <PUB_ID>, <PUB_INT>)
PUB_NAME := publish
PUB_TYPE := built-in
PUB_ID := publish
PUB_INT := (<PUB_MIN_PUBLISHER>, <PUB_MAX_PUBLISHER>)
PUB_MIN_PUBLISHER:= 1
PUB_MAX_PUBLISHER:= 1
```

```
QUEUE := (<QUEUE_NAME>, <QUEUE_TYPE>, <QUEUE_ID>,
    <QUEUE_INT>)
QUEUE_NAME := queue
QUEUE_TYPE := built-in
QUEUE_ID := queue
QUEUE_INT := (<QUEUE_MIN_SOURCE>, <QUEUE_MAX_SOURCE>,
    <QUEUE_MIN_DEST>, <QUEUE_MAX_DEST>)
QUEUE_MIN_SOURCE := 1
QUEUE_MAX_SOURCE := 1
QUEUE_MIN_DEST := 1
QUEUE_MAX_DEST := 1

QUEUE_ALL := (<QUEUE_ALL_NAME>, <QUEUE_ALL_TYPE>,
    <QUEUE_ALL_ID>, <QUEUE_ALL_INT>)
QUEUE_ALL_NAME := queue_all
QUEUE_ALL_TYPE := built-in
QUEUE_ALL_ID := queue_all
QUEUE_ALL_INT := (<QUEUE_ALL_MIN_SOURCE>,
    <QUEUE_ALL_MAX_SOURCE>, <QUEUE_ALL_MIN_DEST>,
    <QUEUE_ALL_MAX_DEST>)
QUEUE_ALL_MIN_SOURCE := 1
QUEUE_ALL_MAX_SOURCE := 1
QUEUE_ALL_MIN_DEST := 2
QUEUE_ALL_MAX_DEST := 6

READ := (<READ_NAME>, <READ_TYPE>, <READ_ID>, <READ_INT>)
READ_NAME := read
READ_TYPE := built-in
READ_ID := read
READ_INT := (<READ_MIN_CLIENT>, <READ_MAX_CLIENT>,
    <READ_MIN_REPOSITORY>, <READ_MAX_REPOSITORY>)
```

```
READ_MIN_CLIENT := 1
READ_MAX_CLIENT := 1
READ_MIN_REPOSITORY := 1
READ_MAX_REPOSITORY := 1

RENDEZ-VOUS := (<RENDEZ-VOUS_NAME>, <RENDEZ-VOUS_TYPE>,
    <RENDEZ-VOUS_ID>, <RENDEZ-VOUS_INT>)
RENDEZ-VOUS_NAME := rendez-vous
RENDEZ-VOUS_TYPE := built-in
RENDEZ-VOUS_ID := rendez-vous
RENDEZ-VOUS_INT := (<RENDEZ-VOUS_MIN_SOURCE>, <RENDEZ-
    VOUS_MAX_SOURCE>, <RENDEZ-VOUS_MIN_DEST>, <RENDEZ-
    VOUS_MAX_DEST>)
RENDEZ-VOUS_MIN_SOURCE := 1
RENDEZ-VOUS_MAX_SOURCE := 1
RENDEZ-VOUS_MIN_DEST := 1
RENDEZ-VOUS_MAX_DEST := 1

SUBSCRIBE := (<SUBSCRIBE_NAME>, <SUBSCRIBE_TYPE>,
    <SUBSCRIBE_ID>, <SUBSCRIBE_INT>)
SUBSCRIBE_NAME := subscribe
SUBSCRIBE_TYPE := built-in
SUBSCRIBE_ID := subscribe
SUBSCRIBE_INT := (<SUBSCRIBE_MIN_SUBSCRIBER>,
    <SUBSCRIBE_MAX_SUBSCRIBER>)
SUBSCRIBE_MIN_SUBSCRIBER:= 1
SUBSCRIBE_MAX_SUBSCRIBER:= 1

WRITE := (<WRITE_NAME>, <WRITE_TYPE>, <WRITE_ID>,
    <WRITE_INT>)
WRITE_NAME := write
WRITE_TYPE := built-in
```

```
WRITE_ID := write
WRITE_INT := (<WRITE_MIN_CLIENT>, <WRITE_MAX_CLIENT>,
    <WRITE_MIN_REPOSITORY>, <WRITE_MAX_REPOSITORY>)
WRITE_MIN_CLIENT := 1
WRITE_MAX_CLIENT := 1
WRITE_MIN_REPOSITORY := 1
WRITE_MAX_REPOSITORY := 1

USER-DEF_CONN := (<UD_CONN_NAME>, <UD_CONN_TYPE>,
    <UD_CONN_ID>, <UD_CONN_INT>)
UD_CONN_NAME := <STRING>
UD_CONN_TYPE := user-def
UD_CONN_ID := <STRING>
UD_CONN_INT := {<UD_CONN_ROLENAME>, <UD_CONN_MIN>,
    <UD_CONN_MAX>}
UD_CONN_ROLENAME := <STRING>
UD_CONN_MIN := <NUM>
UD_CONN_MAX := <NUM>

STRING := {<LETRA> | <NUM>}*
LETRA := a|b|c|...|x|y|z|A|B|C|...|X|Y|Z
NUM := 0|...|9
```