Modelado de Comportamiento de Conectores de Software a través de Lenguajes Declarativos

Fernando Asteasuain
Dpto. de Tecnología y Administración
Universidad Nacional de Avellaneda
España 350 – Avellaneda- BsAs
CAETI-UAI
fasteasuain@undav.edu.ar

Francisco Tarulla
Deparmento de Computación
FCEyN - UBA
Pabellón I Ciudad Universitaria
Intendente Guiraldes 21610
ftarulla@dc.uba.ar

Abstract

La descripción del comportamiento arquitectónico de un sistema requiere contar con una notación expresiva y flexible para poder explorar y razonar sobre las distintas alternativas posibles evitando tomar decisiones prematuras. En este trabajo se explora FVS, un lenguaje declarativo basado en escenarios gráficos, como una posible notación para describir el comportamiento arquitectónico. En particular, el trabajo se enfoca en la especificación de conectores de software. Los resultados obtenidos permiten ilustrar el potencial del lenguaje en el dominio de arquitecturas de software.

1. Introducción

La especificación del comportamiento esperado de un sistema en etapas tempranas es una actividad cada vez más relevante para el desarrollo de software [1-2]. En particular, dado que en etapas iniciales todavía los requerimientos no están maduros es fundamental tener la habilidad de explorar el comportamiento, analizando y razonando distintas variantes para poder elegir la más adecuada, y así evitar lo más posible tomar decisiones prematuras más cercanas a la implementación.

Una de las áreas donde esta exploración del comportamiento es crucial es el diseño de Arquitecturas de Software. Una arquitectura se software se enfoca en describir el comportamiento de los principales componentes del sistema a desarrollar y la manera en que éstos interactúan unos con otros utilizando un gran nivel de abstracción para dejar de lado detalles implementativos de bajo nivel [3-5]. Desde este punto de vista el diseño de arquitecturas de software puede considerarse como el puente que une los requerimientos con la especificación [3].

Existen numerosas alternativas para describir el comportamiento arquitectónico [6-8]. Sin embargo, todavía existen algunos problemas por resolver [8-11]. Uno de los principales desafíos en este contexto se concentra en la notación utilizada para la especificación del comportamiento, la cual debe ser lo suficientemente expresiva y flexible como para poder razonar y analizar distintas opciones. Por ejemplo, un conector complejo como el *Publish/Subcribe*[5] puede configurarse para comportarse de diversas maneras: con o sin time-out, diferentes tipos de protocolos para publicar, distintas formas de comunicación con los suscriptores, etc. La notación utilizada para describir su comportamiento debe poder brindarle la posibilidad al arquitecto de software de analizar y comparar todas las variantes. Muchas veces cumplir tales objetivos puede ser difícil de lograr a través de notaciones operacionales como autómatas o de lenguajes formales de descripción arquitectónica (ADL's) [11-14]. Otro problema relacionado detectado es que en algunos contextos se pasa directamente de la especificación a la implementación [11]. Este quizás anticipado pasaje a código dificulta la exploración del comportamiento.

Las notaciones declarativas son particularmente útiles para describir el comportamiento de sistemas en etapas iniciales ya que las especificaciones resultan cercanas a la manera en que los requerimientos son expresados en lenguaje natural [9]. A modo de ejemplo se pueden mencionar que las notaciones declarativas son ampliamente utilizadas para describir las propiedades que un sistema debe satisfacer en técnicas de validación formal como Model Checking [15]. Dadas estas características, este trabajo se propone explorar notaciones declarativas para la especificación del comportamiento arquitectónico como una alternativa para lograr especificaciones adecuadas para la exploración y el razonamiento en etapas tempranas.

Dentro del comportamiento arquitectónico este trabajo se concentra en el modelado de conectores de software arquitectónicos. Los conectores de software juegan un papel fundamental ya que dictaminan **cómo** y **de qué manera** dos o más componentes de software se comunican entre sí. Es decir, un conector de software establece el protocolo de comunicación entre los componentes de software que interactúan entre sí. Dentro de las posibles vistas arquitectónicas los conectores son considerados ciudadanos de primer mundo en las vistas enfocadas en reflejar el comportamiento dinámico. De tales vistas la más conocida lleva el nombre de vista de "Componentes y Conectores" [16].

El lenguaje declarativo elegido en este trabajo es **FVS** (Feather Weight Visual Scenarios) [17-18]. FVS es un lenguaje declarativo gráfico basado en escenarios utilizado para denotar el comportamiento de sistemas en etapas tempranas. En [17] FVS fue comparado con otras notaciones para modelar patrones de especificación [21], y resultó con especificaciones más compactas, fáciles de comparar y modificar. La utilización de FVS en dominios arquitectónicos fue explorado en [19]. En [19] se utilizó FVS para modelar la arquitectura de familia de productos. En el presente trabajo se profundiza y extiende este trabajo buscando consolidar al lenguaje FVS como un lenguaje para la especificación del comportamiento arquitectónico. Se modelan conectores de mayor complejidad mostrando la posibilidad de razonar sobre distintas alternativas de comportamiento. Asimismo, se estudian dos casos de aplicaciones concretas como ejemplos de aplicabilidad del lenguaje.

El resto del presente trabajo se estructura de la siguiente manera. La sección 2 presenta informalmente al lenguaje FVS. En la sección 3 se modela en FVS el comportamiento del conector *Pipe* y una posible variante. En la sección 4 se modelan en FVS dos casos de estudio que reflejan el comportamiento de conectores de software en aplicaciones concretas. La sección 5 presenta trabajo relacionado y futuro mientras que la sección 6 exhibe las conclusiones.

2. FVS: Feather Weight Visual Scenarios

En esta sección se describirán de manera informal las principales características de FVS. Una caracterización formal del lenguaje se puede ver en [17-18].

FVS es un lenguaje gráfico basado en escenarios. Los escenarios son órdenes parciales de eventos basados en puntos, los cuales son etiquetados con los posibles eventos que pueden ocurrir en ese punto, y en flechas que los conectan. Una flecha entre dos puntos indica precedencia del origen respecto del destino. Por ejemplo, en la Figura 1-a el evento A precede al evento B, indicando que ocurre antes. FVS cuenta con abreviaciones para indicar la próxima ocurrencia de un evento luego de otro, o la ocurrencia inmediata anterior de un evento precediendo a otra. Para el primer caso, la abreviación se denota gráficamente con una segunda flecha (abierta) cerca del punto de destino. Por ejemplo, en la Figura 1-b el escenario captura la primera ocurrencia de un evento B que sigue a un evento A. Para el segundo caso, la segunda flecha se ubica cerca del punto de origen. Continuando el ejemplo, el escenario en la Figura 1-c captura la ocurrencia inmediata anterior de un evento A que precede a un evento B.

Se introducen en la notación dos puntos destacados para denotar el principio y el fin de una ejecución: un círculo completo grande para el comienzo y dos círculos concéntricos para el final (ilustrado en la Figura 1-d). Las flechas pueden ser etiquetadas para restringir comportamiento, interpretándose como eventos prohibidos. En la Figura 1-e el evento A precede al evento B de manera tal que el evento C no ocurre entre ambos. Notar en este punto que los operadores de eventos próximo y anterior introducidos previamente pueden modelarse con restricciones como se ve en la Figura 1-f. El escenario en la parte superior de la Figura 1-f es equivalente al escenario en la Figura 1-b, y similarmente lo son el escenario en la parte inferior de la Figura 1-f con el escenario en la Figura 1-c. Finalmente, FVS cuenta con la posibilidad de definir "aliasing" entre puntos. El escenario en la Figura 1-g indica que la ocurrencia del evento A también implica la ocurrencia simultánea del evento B. En este punto vale la pena destacar que el evento A se repite en el etiquetado del segundo punto sólo para una cuestión formal de la sintaxis del lenguaje [17-18].

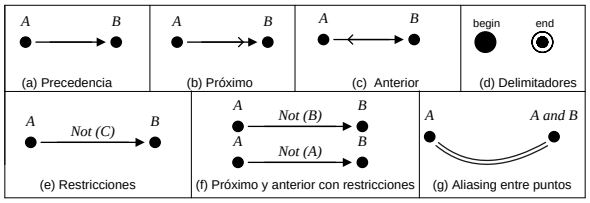

Figura 1. Elementos Básicos de FVS

2.1. Reglas FVS

En esta sección se presenta el concepto de Reglas FVS, un concepto clave del lenguaje. Básicamente, una regla está dividida en dos partes: un escenario tomando el rol del antecedente, y al menos un escenario tomando el rol de un consecuente. La intuición es que siempre que una traza "encuentre" un antecedente, entonces también debe encontrar al menos uno de los consecuentes. En otras palabras, una regla toma la forma de una implicación, con un escenario antecedente, y uno o más escenarios consecuentes.

El escenario antecedente es una sub-estructura común a todos los consecuentes, lo cual permite establecer relaciones complejas entre los puntos del antecedente y consecuentes. Esto otorga gran flexibilidad a FVS, ya que sus reglas no están limitadas, como la mayoría de las aproximaciones que manejan algún tipo de escenarios de implicación, donde el antecedente funciona únicamente como una estructura que debe preceder a los consecuentes. De esta manera las reglas pueden expresar comportamiento que ocurrió en el pasado, o en el medio de otros eventos. Gráficamente, el antecedente se muestra de color negro, mientras que los consecuentes en color gris. Como las reglas pueden tener más de un consecuente, cada elemento que no pertenece al antecedente se identifica con un número para identificar a qué consecuente pertenece. La semántica de FVS estará dada por el conjunto de trazas que satisfaga el conjunto de reglas describiendo el comportamiento del sistema.

La Figura 2 muestra un ejemplo de una regla FVS. La misma modela parte del comportamiento de un conector de software *Call Sincrónico* con timeout [5]. La regla establece que entre dos llamadas consecutivas del componente en el rol del "llamador" tienen que haber ocurrido una de situaciones posibles: o bien se obtuvo una respuesta del componente en el rol del "llamado" (consecuente 1) o bien ocurrió un timeout (consecuente 2)

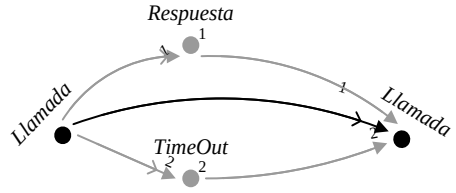

Figura 2. Ejemplo de una regla en FVS

3. Modelado de Conectores en FVS

En esta sección se modela en FVS parte del comportamiento del conector *Pipe*. La sección 3.1 modela una posible variante de este conector y la sección 3.2 presenta algunas observaciones realizadas y una breve comparación con otra notación.

En pocas palabras, un *pipe* es una estructura intermedia que conecta el flujo de comunicación entre dos componentes. En general, toma el dato de la salida de un componente y lo ubica como dato de entrada de otro. Suele formar parte de una estructura más compleja de componentes que van transformando datos de manera secuencial representando un estilo arquitectónico clásico denominado *Pipe and Filter* [5,16]. A continuación se modela parte del comportamiento de un conector Pipe, tal como es especificado en [5].

En [5] se describe un *Pipe* como una estructura intermedia al estilo de un repositorio, donde un componente "productor" escribirá datos en la estructura, mientras que otro "consumidor" se encargará de leerlos, siguiendo siempre un protocolo establecido por el comportamiento del conector. Los siguientes puntos describen parte del comportamiento de este Pipe:

1. Como toda estructura intermedia el Pipe debe abrirse antes de su utilización y cerrarse al finalizar.
2. El protocolo del componente productor y consumidor incluye un evento de "Fin de Escritura" y "Fin de Lectura", indicando el fin de su interacción con el pipe.
3. Ante cada evento de "Fin de Escritura" el pipe debe incluir un evento de "Fin de Archivo", a modo de indicación para el consumidor.
4. Una vez que se encuentra "Fin de Archivo" la única acción posible para lectura del Pipe es un "Fin de Lectura".

La figura 3 muestra dos reglas FVS describiendo el comportamiento general del pipe tomando lo especificado en el requerimiento 1 y parte del 2. La regla de la izquierda establece que todo evento "Pipe_Abierto" debe ser seguido de un evento "Pipe_Cerrando", validando que la estructura quede cerrada al finalizar su uso. La regla de la derecha indica que siempre que un pipe ha sido abierto y posteriormente cerrado, entonces durante ese lapso deben haber ocurrido dos eventos: la finalización de escritura por parte del consumidor (evento Fin_Escritura) y la finalización de lectura por parte del consumidor (evento Fin_Lectura). Esta regla busca asegurar que el productor y el consumidor han cumplido con su parte del protocolo. Notar que la regla no impone un orden de ocurrencia entre los eventos "Fin_Lectura" y "Fin_Escritura". Es decir, puede tanto terminar primero de escribir el productor como de leer el consumidor, respetando lo especificado en el protocolo. Esto demuestra la flexibilidad de la notación FVS.

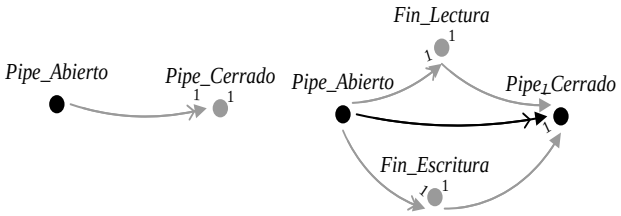

Figura 3. Reglas del Comportamiento General del Pipe

La figura 4 muestra cuatro reglas para modelar la interacción del pipe con los componentes productor y consumidor. En particular, se enfocan en los eventos "Fin_Escritura" y Fin_Lectura", tal como es descrito en el segundo requerimiento presentado anteriormente. Las dos reglas sobre la izquierda controlan que no hay una

nueva interacción con el pipe una vez que los componentes productor o consumidor han indicando el fin de su interacción con el pipe. En cambio, las reglas de la derecha establecen que siempre que haya sucedido una interacción con el pipe (evento Escritura o Lectura) entonces debe haber sucedido en el pasado que el pipe haya sido abierto y que desde entonces no haya sucedido el evento de fin de interacción con el pipe correspondiente (evento Fin_Escritura o evento Fin_Lectura).

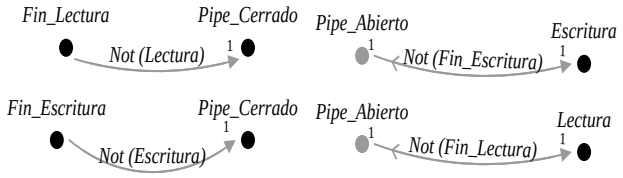

Figura 4. Reglas para el comportamiento interno del Pipe

Finalmente, las reglas de la figura 5 se concentran en los requerimientos 3 y 4. La regla de la izquierda establece que cada evento "Fin_Escritura" debe ocurrir simultáneamente con un evento "Fin_Archivo" (requerimiento 3). La regla de la derecha valida que no ocurran lecturas sobre el pipe una vez que se encontró el evento "Fin_Archivo" hasta la ocurrencia del evento "Fin_Lectura" (requerimiento 4).

Figura 5. Reglas relativas al evento Fin de Archivo

3.1. Modelado de una variante del Pipe

En esta sección se modela el comportamiento de una variante del conector presentado en la sección anterior. En esta variante el pipe tiene una estructura limitada para almacenamiento, por lo que el productor no puede escribir en el pipe mientras el mismo esté completo. La figura 6 refleja dos reglas modelando esta restricción. La regla sobre la izquierda dice que si ocurre un evento *Pipe_Lleno* y luego en el futuro ocurre un evento de escritura, entonces tiene que haber sucedido durante ese lapso un evento de escritura por parte del consumidor liberando un lugar en la estructura y además, que nunca haya vuelto a llenarse el pipe. La figura de la derecha tiene dos posibles consecuentes indicando dos situaciones válidas para que ocurra una interacción de lectura con el pipe. O bien nunca se llenó el pipe desde que fue abierto (consecuente 1) , o si ocurrió, entonces tiene que haber sucedido una lectura sin que vuelva a llenarse (consecuente 2).

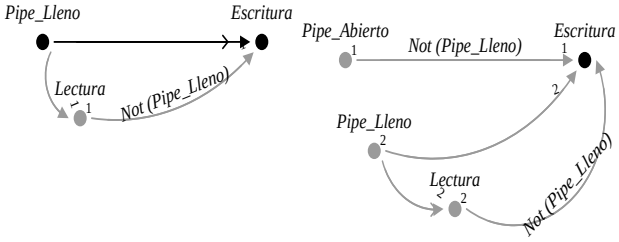

Figura 6. Reglas para un Pipe con tamaño limitado.

3.2. Algunas observaciones

La figura 7 muestra el comportamiento del conector pipe utilizando una notación basada en autómatas finitos etiquetados, obtenida de [5].

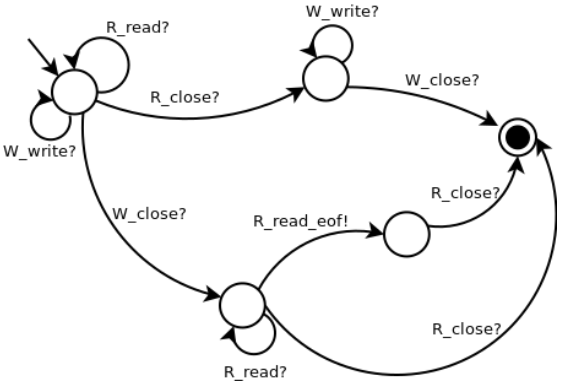

Figura 7. Comportamiento del Pipe con Autómatas.

Los eventos marcados con signo de pregunta (?) simbolizan datos producidos en otro autómata, mientras que aquellos con signo de exclamación (!) son producidos en el autómata. De esta manera se realiza la sincronización entre distintos autómatas. Esto implica que en muchas ocasiones para comprender, analizar o introducir cambios en el comportamiento del conector es necesario analizar varios autómatas y la manera en que se componen y sincronizan. Por ejemplo, para introducir el cambio propuesto en la sección 3.2, el autómata de la figura 7 tendría que ser analizado junto a los autómatas que describen el comportamiento del productor y del consumidor. Esto podría ser una posible dificultad a la hora de introducir modificaciones y analizar y explorar distintas posibilidades de comportamiento arquitectónico. En las reglas FVS los cambios fueron realizados agregando nuevas reglas, adaptando únicamente las condiciones válidas para realizar una escritura. Dichos cambios se corresponden con el cambio expresado en lenguaje natural que buscaba evitar escrituras cuando el

pipe estuviese lleno. Si bien estudios más profundos son necesarios para confirmar la flexiblidad de la notación, el ejemplo permite ilustrar el potencial de la misma para explorar de manera intuitiva distintas posibilidades de comportamiento, característica fundamental en la etapa de diseño de la arquitectura.

Una observación final es notar cómo se modela el requerimiento relativo al evento Fin de Archivo que debe ocurrir una vez que el productor finaliza su escritura. Mientras que en FVS tal funcionalidad está explícitamente modelada (ver Figura 5) en el autómata de la Figura 7 esta modelado con una única transición que forma parte de un autómata complejo, con lo cual puede ser no trivial encontrar dónde y cómo está modelado esta parte del protocolo del conector. De ser necesario introducir cambios o contemplar otras posibilidades de comportamiento, el hecho de que esté modelado explícitamente y de manera localizada como en FVS puede ser un factor que minimice el impacto del cambio y favorezca la exploración del comportamiento.

4. Casos de Estudio

En esta sección se modela el comportamiento de dos conectores utilizados en ejemplos concretos obtenidos de la literatura [11,22]. En la sección 4.1 se muestra un conector destinado a proveer balance de carga dentro de un servidor. En la sección 4.2 se modela un conector destinado a brindar interacción entre una aplicación y la plataforma de microblogging *Twitter*.

4.1. Conector Balance de Carga

En esta sección se modela un conector de balance de carga basado en el caso de estudio presentado en [11]. Una manera clásica de aumentar la disponibilidad de un sistema es introduciendo múltiples servidores y aplicando una técnica de balance de carga. Dicha técnica distribuye equitativamente los pedidos de los clientes entre los posibles servidores. En [11] se tomó la decisión de incluir esta responsabilidad a través de un conector de balance de carga que haga de intermediario entre los clientes y los servidores.

Dicho conector funciona bajo dos modalidades excluyentes: adaptiva y no adaptiva. La primera modalidad incluye en el criterio de distribución la carga de los servidores. Ejemplos de métricas posibles son el número de pedidos o la cantidad de ciclos ociosos de CPU. Las técnicas no adaptivas no tienen en cuenta la carga de cada servidor. Un ejemplo clásico es una distribución *Round Robin* entre los servidores.

Las reglas de la figura 8 ilustran parte del comportamiento del conector bajo la modalidad adaptiva donde se usa como métrica la cantidad de ciclos ociosos de CPU. Cualquier servidor que supere un cierto umbral de ciclos ociosos se asumirá listo para ejecutarse. Para la especificación del comportamiento se asume un sistema con dos servidores y eventos "Ocioso_S_i", para denotar que el servidor S_i se encuentra ocioso y por lo tanto disponible para tomar pedidos, "Ocupado_S_i", para denotar que el servidor S_i se encuentra ocupado y "Servidor-*i*" para representar que el servidor S_i ha recibido un pedido. Las dos reglas en la parte superior de la figura 8 indican que cada servidor tomará pedidos únicamente si se encuentran ociosos. La regla en la parte inferior dice que en caso de estar ambos servidores libres, cualquiera de los dos puede tomar el próximo pedido. El servidor 1 es activado en el consecuente 1, mientras que el servidor 2 se activa en el consecuente 2.

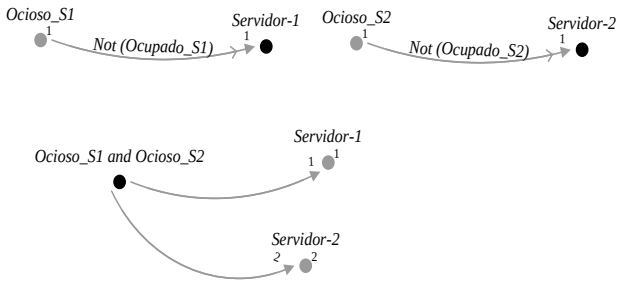

Figura 8. Reglas FVS – Balance de Carga Adaptivo.

Las reglas de la figura 9 modelan parte del comportamiento del conector bajo la modalidad de *Round Robin*. La regla de la izquierda establece que entre dos pedidos consecutivos tomados desde el servidor 1, el servidor 2 tiene que haber tomado un pedido. Análogamente, la regla de la derecha establece que entre dos pedidos consecutivos asignados al servidor 2 se tiene que haber asignado un pedido al servidor 1.

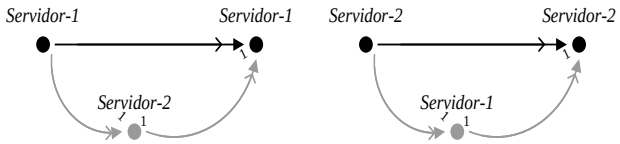

Figura 9. Reglas FVS – Balance de Carga No Adaptivo.

4.2. Conector Interacción Aplicación - Twitter

En esta sección se especifica el comportamiento de un conector que realizar la interacción entre una aplicación ficticia y la red social Twitter. El mismo está basado en el ejemplo presentado en [22].

La aplicación a desarrollar interactúa con la red social *Twitter*. Dicha red social provee una interfaz de programación (API) cubriendo las siguientes responsabilidades [20]:

1. Un usuario puede enviar un *Tweet* para actualizar su status.
2. Los usuarios recibirán los tweets de aquellas personas con las cuales estén en contacto.
3. Los usuarios pueden buscar tweets usando tópicos predefinidos denominados "hashtags".
4. Para realizar cualquiera de los tres servicios anteriores, el usuario debe autenticarse.

El conector a desarrollar debe coordinar y comunicar la aplicación y la API de *Twitter*. Esto implica que su funcionalidad debe incluir los cuatro servicios mencionados. Siguiendo la decisión tomada en [22], se especifica un único conector que cubra los cuatro servicios en vez de cuatro conectores por separado para cada uno de los servicios.

Los requerimientos del conector que serán modelados a continuación son los siguientes. En primer término, la **autenticación**. Antes de realizar cualquier interacción (enviar, recibir, o buscar tweets) el usuario debe estar autenticado). El segundo requerimiento describe el **envío de datos** de los tweets. El envío de datos de Tweets puede tomar únicamente dos formas: envío de texto (envío plano) o envío multimedia, según el tweet contenga o no fotos o videos. Finalmente, el último requerimiento cubre el aspecto de **compresión** de datos. Todos los tweets serán comprimidos desde el conector una vez que son recibidos desde Twitter, y serán descomprimidos nuevamente antes de ser enviados a la aplicación.

Las reglas de la Figura 10 modelan los mencionados requerimientos. La regla en la Figura 10-a valida que cualquier invocación de un servicio haya sido realizada por un usuario autenticado. Las reglas en la Figura 10-b se concentran en el envío de datos. La regla de la izquierda controla que todo envío de un tweet será o bien un envío multimedial (consecuente 1) o un envío plano (consecuente 2). La regla de la parte derecha controla que previo a todo envío multimedial el usuario haya cargado previamente algún video o imagen, descartando que se haya usado en algún tweet de tipo plano. Finalmente, la regla de la Figura 10-c establece que todo tweet recibido desde *Twitter* sea comprimido al llegar al conector, y luego, que sea descomprimido al ser enviado a la aplicación.

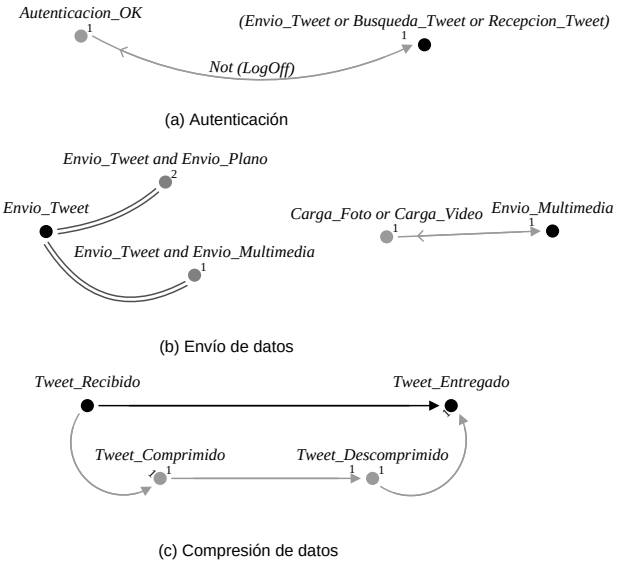

Figura 10. Reglas Conector Aplicación - Twitter

5. Trabajo Relacionado & Futuro

Property Sequence Chart (PSC) [23] es otro lenguaje gráfico utilizado para especificar comportamiento de components arquitectónicos [6]. Está inspirado en diagramas de UML 2.0 (Interaction Sequence Diagrams). Con respecto a FVS existen algunas diferencias como lenguajes visuales. En PSC describir restricciones complejas de comportamiento puede llegar a necesitar el uso extra de anotaciones en modo texto. Otra diferencia radica en el hecho que las propiedades de comportamiento en PSC se describen como anti-escenarios mientras que en FVS el comportamiento está basado en reglas.

Entre otras aproximaciones posibles se pueden mencionar aquellas basadas en lenguajes formales de descripción arquitectónica ADL's (Architectural Description Languages) [12-14]. Sin embargo, no contar con una descripción visual del comportamiento podría ser una desventaja a la hora de explorar y razonar sobre el comportamiento [24].

Finalmente también se puede mencionar el trabajo [25], donde utilizan el lenguaje visual VTS [26] para modelar comportamiento arquitectónico utilizando redes de Petri. El lenguaje FVS es un lenguaje basado en VTS, pero mucho más simple (no cuenta, por ejemplo con constructores temporales, entre otras diferencias) y rediseñado para describir comportamiento de sistemas en vez de propiedades [cita 11]. Mientras que en [25] el formalismo utilizado en redes de Petri en el presente trabajo todo el comportamiento se basa en reglas FVS.

Respecto del trabajo futuro hay varias líneas que pretenden continuar esta exploración inicial de FVS en

dominios arquitectónicos presentada en este trabajo. Por un lado, dado que las reglas FVS pueden traducirse en autómatas de Buchi [18] sería interesante poder analizar la combinación de FVS con otros lenguajes de descripción arquitectónicas como los mencionados en el segundo párrafo de esta sección así como también con otros. También se podría dar un paso más validando formalmente propiedades arquitectónicas utilizando Model Checkers [6].

Se pretende describir más conectores típicos de software como para consolidar a FVS como un lenguaje para describir comportamiento arquitectónico. En este sentido, también es necesario realizar una comparación más profunda de FVS contra otras notaciones como máquinas de estado, para poder tener conclusiones formales y más robustas sobre la flexibilidad y poder expresivo de la notación.

6. Conclusiones

En el presente trabajo se presenta FVS como un lenguaje gráfico declarativo para especificar el comportamiento arquitectónico. En particular, el trabajo está enfocado en el modelado del comportamiento de conectores de software. Para mostrar la flexibilidad y poder expresivo de la notación se especificó un conector típico arquitectónico como es el conector *Pipe* junto con una posible variante del mismo. Adicionalmente, se incluye la especificación de dos conectores complejos utilizados en situaciones específicas: para el balance de carga en un servidor, y para realizar la interacción entre una aplicación y la red social Twitter.

Si bien nuevos estudios son necesarios para consolidar y validar los resultados obtenidos, los mismos son suficientes para mostrar el potencial de FVS como lenguaje de descripción arquitectónica, en especial para la exploración y el razonamiento sobre el comportamiento en etapas iniciales.

7. Agradecimientos

Este trabajo fue financiado por Proyecto UNDAVCYT 2014, PROYECTO CAETI - Universidad Abierta Interamericana (UAI).

Referencias

[1] Horkoff, J., Maiden, N., & Lockerbie, J. Creativity and goal modeling for software requirements engineering. In Proceedings of the 2015 ACM SIGCHI Conference on Creativity and Cognition (pp. 165-168). ACM. (2015, June)

[2] Braude, E. J., & Bernstein, M. E. Software engineering: modern approaches. Waveland Press. (2016).

[3] Garlan, D. (2003). Formal modeling and analysis of software architecture: Components, connectors, and events. In Formal Methods for Software Architectures (pp. 1-24). Springer Berlin Heidelberg.

[4] Bass, L. (2007). Software architecture in practice. Pearson Education India.

[5] Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2009). Software architecture: foundations, theory, and practice. Wiley Publishing.

[6] Pelliccione, P., Inverardi, P., & Muccini, H. Charmy: A framework for designing and verifying architectural specifications. TSE , 35(3), 325-346. (2009)

[7] Chen, L., Huang, L., Zhong, H., Li, C., & Wu, X. Breeze: A modeling tool for designing, analyzing, and improving software architecture. IEEE 23rd RE (pp. 284-285). IEEE. (2015).

[8] Aldrich, J., Chambers, C., & Notkin, D. ArchJava: connecting software architecture to implementation. ICSE 2002 (pp. 187-197). (2002).

[9] Van Lamsweerde, A. (2003). From system goals to software architecture. In Formal Methods for Software Architectures (pp. 25-43). Springer Berlin Heidelberg

[10] Albin, S. T. The art of software architecture: design methods and techniques (Vol. 9). John Wiley & Sons (2003).

[11] Matougui, S., & Beugnard, A. (2005, June). How to implement software connectors? a reusable, abstract and adaptable connector. In IFIP International Conference on Distributed Applications and Interoperable Systems (pp. 83-94). Springer Berlin Heidelberg.

[12] Chen, L., Huang, L., Zhong, H., Li, C., & Wu, X. Breeze: A modeling tool for designing, analyzing, and improving software architecture. IEEE 23rd RE (pp. 284-285). IEEE. (2015).

[13] Oquendo, F., Leite, J., & Batista, T. Specifying Architecture Behavior with SysADL. In 2016 13th Working IEEE/ (WICSA) (pp. 140-145). IEEE (2016)

[14] Blom, H., Chen, D. J., Kaijser, H., Lönn, H., Papadopoulos, Y., Reiser, M. O., ... & Tucci, S. EAST-ADL: An Architecture Description Language for Automotive Software-intensive Systems in the Light of Recent use and Research. (IJSDA),5(3), 1-20 (2016)

[15] Clarke, E. M., Grumberg, O., & Peled, D. (1999). Model checking. MIT press.

[16] Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., & Little, R. (2002). Documenting software architectures: views and beyond. Pearson Education.

[17] Asteasuain, F., Braberman, V. Specification patterns: formal and easy. IJSEKE. Print ISSN: 0218-1940. 25(4) 669-700. (2015).

[18] Asteasuain, F., Braberman,V. Declaratively building behavior by means of scenario clauses. RE journal, DOI 10.1007/s00766-015-0242-2. ISSN: 0947-3602 (2016).

[19] Asteasuain, F., Vultaggio, L. P. Declarative and Flexible Modeling of Software Product Line Architectures. IEEE Latin America Transactions, 14(2), 885-892 (2016).

[20] Honey, C., & Herring, S. C. (2009, January). Beyond microblogging: Conversation and collaboration via Twitter. In System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on (pp. 1-10). IEEE.

[21] Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1999, May). Patterns in property specifications for finite-state verification. In Software Engineering, 1999. Proceedings of the 1999 International Conference on (pp. 411-420). IEEE.

[22] Slotos, T. (2014, June). A specification schema for software connectors. In Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering (pp. 139-148). ACM.

[23] Autili, M., Inverardi, P., & Pelliccione, P. A scenario based notation for specifying temporal properties. In Proceedings of the SCESM workshop s (pp. 21-28). ACM. (2006).

[24] Holzmann, G. J. (2002, November). The logic of bugs. In Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering (pp. 81-87). ACM.

[25] D. Monteverde, A. Olivero, S. Yovine, and V. Braberman. VTS-based Specification and Verification of Behavioral Properties of AADL Models. In (ACES'08). (2008).

[26] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual Timed Event Scenarios. In Proc. Of the 26th ACM/IEEE ICSE '04. ACM Press (2004).