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

Tesis de Licenciatura

Refinamiento arquitectónico basado en estilos: conceptualización y aplicación

Autor:

- Martín Nicolás Burak
  LU: 57/02

Directores:

- Dr. Victor Braberman
- Dr. Sebastián Uchitel

Resumen

La arquitectura de software es un área de estudio relativamente nueva que estudia las estructuras del software y sus propiedades externamente visibles. Mientras que otros aspectos del área se han desarrollado y consolidado en forma definitiva, el concepto de refinamiento arquitectónico no ha sido estudiado ni desarrollado extensivamente. En este trabajo se presenta un marco teórico-formal para el estudio del refinamiento arquitectónico. A partir del mismo, se provee una nueva definición de refinamiento basada en estilos arquitectónicos y en la preservación parcial de propiedades. Se desarrolla luego, una metodología de refinamiento con estilos utilizando el model checker LTSA; gracias a esta metodología es posible verificar automáticamente si dos descripciones arquitectónicas (expresadas mediante las herramientas provistas por LTSA) configuran un refinamiento válido. Se ilustra el uso de la metodología propuesta mediante diferentes ejemplos y se comprueba que la misma permite validar refinamientos de naturaleza compleja. Finalmente se diserta sobre los roles que una metodología como la presentada puede jugar en el desarrollo de sistemas y sobre los beneficios que supone su uso.

Agradecimientos

Estos agradecimientos se comenzaron a escribir, imaginariamente, mucho antes que cualquier otra sección del trabajo. Es una gran satisfacción personal saber que todas las personas que me imaginaba iban a figurar aquí, efectivamente lo hacen. Esta es una de las tantas cosas que me confirman, día a día, que he sabido elegir a la gente de la cual me rodeo y que la vida me ha provisto maravillosamente del resto. Sin más rodeos, la gran lista de agradecimientos:

A mis directores Víctor y Sebastián que me han guiado sabiamente y han sabido soportar mis delirios durante más de un año de desarrollo. A los jurados Mariano y Santiago que han aportado su invaluable y totalmente desinteresada colaboración. A Diego y Nicolás que me han contactado con mis directores y me ayudaron, en su rol de jefes, a conocer el Departamento y su gente. Al Departamento de Computación, a la Facultad de Ciencias Exactas y Naturales y a la UBA toda como instituciones de excelencia de desarrollo social. A toda la gente de secretaría: Nicolás, Mercedes, Aida e Ignacio cuyo trabajo permite el desarrollo de todas las actividades en el Departamento. A la vieja guardia de administradores: Fer, Gabo, Ale, Maxi, Cristian y Juan que además de haber sido compañeros de trabajo son, más que nada y en presente, amigos. A los pibes que siempre están ahí, no es necesario que los liste, ellos saben quiénes son; verdaderos amigos, aunque sumamente disfuncionales como grupo, que me han acompañado (y soportado los desplantes) durante todos estos año de carrera. A los amigos de la ORT, que aunque ya no son parte del día a día siguen en el corazón. A mis amigos de la vida Iván y Axel, a los que debería llamar más seguido. A mi “familia política” Chechi (perdón, Cecilia), Cesar y Mariana que me han aceptado como uno más en su casa y a mis “amigas políticas” Barbi y Caskis porque así lo prometí. A la familia que no es familia: Mops, Frydman, Libhaher y Kruk, los primos, primas, tíos y tías que me ha regalado la vida y mis padres. A mi zeides, bobes, tíos, tías, primos y primas “de verdad” y por último a mi familia amada: Mamá, Papá, Judi, Andrés, Marcelo y Laura, simplemente gracias.

Agradecimiento

No, esto no es un error en el documento. Lo excepcional debe separarse de lo general y más allá de mi eterno agradecimiento a todas las personas citadas anteriormente, debo ser fiel a los hechos y a los sentimientos. Este agradecimiento, único y especial, es para la más especial y única de todas las mujeres. Para mi nena hermosa, mi gordita adorada, la que me lee como un libro y la que, lamentablemente, se ha llevado la peor parte de este proceso al tener que aguantar cada uno de mis olvidos, desplantes y nervios. A ella que supo responder a todo con su hermosa forma de amar, a ella que me hizo comprender a Saint Exupéry, a ella que es diferente a todas las demás rosas del jardín, para mi “plor” hermosa, para ella todo mi agradecimiento y todo mi amor. Te amo mi gordita hermosa, simplemente gracias.

Tabla de contenidos

1. Introducción

1.1 Presentación del campo

La arquitectura de software es un campo de estudio relativamente nuevo que surge como tal durante la década de 1990, con varios trabajos que hoy pueden considerarse fundacionales [AG92, GS93, GS96, SDK+95]. Si bien varios de los conceptos involucrados en el campo ya han sido previstos por diferentes autores desde la década de 1970 [Par72, Par79, DK76], fue en los 90 que se formó el corpus mínimo de conocimiento necesario para que la arquitectura de software se acuñara un nombre propio. No existe una definición universalmente aceptada del concepto de arquitectura de software, pero el tiempo ha determinado un cierto consenso mínimo al respecto de la misma. En general, se acepta que la arquitectura de software es el estudio de la estructura o estructuras de un sistema, las cuales se componen de un número de elementos que exhiben ciertas propiedades externamente visibles en forma individual (incluido comportamiento) y que se encuentran relacionadas de tal forma que logran exhibir nuevas propiedades a partir de sus relaciones [BCK03].

Actualmente estancado en la etapa de "Trough of Disillusionment" de una hipotética "Hype Curve" de las ciencias de la computación [Hype], el estudio de la arquitectura de software ha logrado inculcar algunos conceptos difusamente definidos en la comunidad no académica. La palabra "arquitectura" y "arquitecto" pasaron a formar parte del folklore del desarrollo de software comercial, despertando odios [Fowler03] y amores por igual. Varias metodologías de desarrollo "modernas" se basan en mayor o menor medida en la existencia del concepto de arquitectura: El Unified Process [JBR99] y sus derivados se dicen "architecture-centered" ya que posicionan a la arquitectura en un rol central como una "perspectiva completa del sistema" a partir del cual "se comprende el proyecto, se organiza el desarrollo, se fomenta la reutilización y se hace evolucionar el sistema". En el marco de la metodología de Extreme Programming [Beck99] existe el concepto de "architectural spike" que da lugar a las "metáforas del sistema" que son "formas de explicar la arquitectura de un sistema para un conjunto de stakeholders" [KBNB04].

De todos los conceptos incluidos en el estudio de la materia, sin dudas el que logró arraigarse con más fuerza es el del "diagrama arquitectónico". Los diagramas arquitectónicos son, a grandes rasgos, representaciones de una porción o de toda la arquitectura de un sistema. Es decir, son una forma de descripción arquitectónica[1]. Generalmente son "diagramas de cajas y líneas" (box-and-line diagrams) ad-hoc que no se apegan a ninguna convención ni cuentan con una semántica formal definida. Aunque pareciera que la definición anterior contradice la aplicabilidad y efectividad de dichos diagramas, estos han sido adoptados y utilizados ampliamente. Este fenómeno puede deberse a:

[1] Cabe aclarar que el uso del término "descripción" no implica que estas se correspondan con un Lenguaje de Descripción Arquitectónico (ADL) dado.

1. Los diagramas arquitectónicos sirven como herramienta de comunicación entre pares (por ejemplo, miembros de un mismo equipo que comparten conocimiento previo)

2. Los diagramas arquitectónicos sirven como herramienta de prototipación muy rudimentaria que permite "pensar" un modelo básico del sistema

3. Los diagramas arquitectónicos son, tal vez, la única forma de presentar el sistema en forma "agradable" a stakeholders no técnicos. La búsqueda de imágenes sobre "software architecture" mediante cualquier buscador de Internet arrojará imágenes coloridas y visualmente muy agradables, con la capacidad de impresionar para bien a la audiencia no técnica (Es decir, son "espejitos de colores")

Más allá de que el punto tres puede provocarle rechazo a cualquier persona que, como el autor, intente definir un marco serio y formal para el estudio de la materia, no se puede ignorar la realidad si se desea obtener resultados aplicables en la práctica. Sea cual fuere el motivo, los diagramas arquitectónicos se han popularizado e instalado de tal forma y con tal fuerza que su uso ya se ha colado en otros ámbitos: El mundo del marketing del software, por ejemplo, parió el concepto de "Marketecture" [Hoh03]. Varios productos de software comerciales incluyen entre sus argumentos de venta su "arquitectura mejorada o superior", por ejemplo el navegador Google Chrome [Chrome] de Google. La recepción de los diagramas arquitectónicos en la práctica fue tal (sobre todo en comparación a otros conceptos de la materia y a otras formas de representación arquitectónica) que mucha gente del medio cree, erróneamente, que un diagrama arquitectónico y la arquitectura del sistema que el diagrama exhibe son lo mismo.

El segundo de los conceptos que ha logrado traspasar la barrera académica, es el de estilos arquitectónicos. Hoy es común escuchar hablar de arquitecturas *Client-Server*, por ejemplo. Los estilos arquitectónicos fueron adoptados para caracterizar y agrupar arquitecturas de acuerdo a sus propiedades. Más allá de que no exista una definición universal de lo que significa que una arquitectura sea *Client-Server*, existe cierto consenso al respecto: una arquitectura *Client-Server* cuenta con un *Servidor* y un número de *Clientes* que realizan llamadas al primero. La comunicación siempre es iniciada por los *Clientes,* mientras que el *Servidor* espera en forma pasiva por sus llamadas. Este consenso es parte del "folklore del desarrollo de software". Sucede lo mismo con otros estilos como *Announcer-Listener* o *Shared Repository* [GS96], todos forman parte de un imaginario colectivo (o "Alucinación compartida" como en [KJ99]) que habilita un intercambio de ideas con mayor facilidad.

Los diagramas arquitectónicos y los estilos arquitectónicos encontraron su lugar en el desarrollo de software como recursos ad-hoc. En general se crean diagramas al comenzar un proyecto para comunicar ciertas ideas de diseño de alto nivel acerca del sistema, apelando al conocimiento "folklórico" de uno o más estilos arquitectónicos. Apegándose a esta forma de trabajo, se presenta un sistema de ejemplo utilizando el siguiente "diagrama arquitectónico":

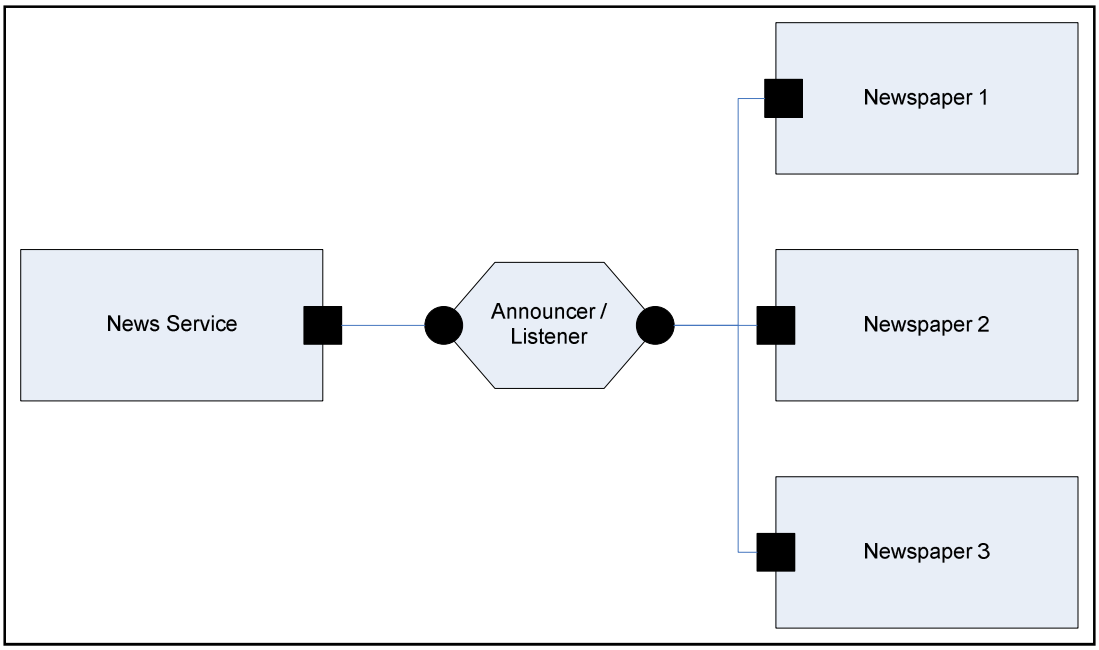

Figura 1-1: Primer diagrama arquitectónico para el sistema agencia de noticias

El diagrama describe la arquitectura de un sistema desarrollado por una agencia de noticias para distribuir boletines de noticias a las redacciones que hayan contratado el servicio. Cada una de las redacciones puede elegir en forma individual si desea suspender la recepción de nuevos boletines (para evitar saturar a sus redactores). Una vez suspendida las recepciones, éstas se pueden reanudar mediante un pedido explicito a la agencia de noticias. Se ha elegido implementar el sistema utilizando el estilo *Announcer-Listener*; la agencia de noticias es el *Announcer*, las redacciones son los *Listeners* y los boletines de noticas son los *Anuncios* o *Eventos*. Hasta aquí, la solución parecería ser "apropiada" de acuerdo al "folklore" actual del desarrollo de software.

Para reducir costos, la comunicación entre las redacciones y la agencia de noticias se realizará a través de Internet. Por cuestiones de seguridad, los servidores de la agencia de noticias se encuentran detrás de un firewall que prohíbe toda comunicación que no corresponda al protocolo HTTP en su puerto estándar. Por su parte, muchas de las redacciones tienen restringidas sus conexiones a Internet mediante firewalls que solo permiten conexiones HTTP de salida y que prohíben cualquier tráfico de entrada. El sistema debería ser capaz de funcionar en este contexto restringido. El problema es que, de acuerdo a la definición "folklórica" del estilo utilizado, la comunicación entre la agencia y las redacciones debe ser iniciada por la primera. Las redacciones son entes pasivos a la espera de *eventos,* pero las restricciones impuestas por los firewalls impiden cualquier comunicación que no sea iniciada desde las mismas. De esta manera, la solución propuesta (es decir, el diagrama) no satisface los requerimientos del sistema y no puede ser implementada. Se propone entonces la siguiente solución:

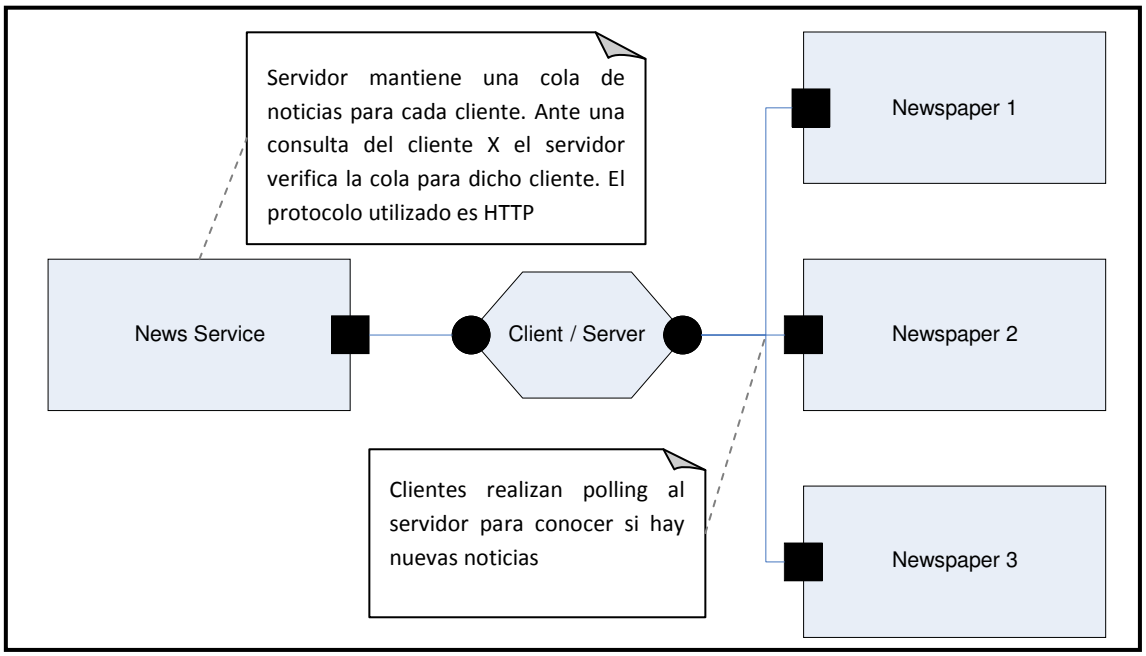

Figura 1-2: Segundo diagrama arquitectónico para el sistema agencia de noticias

Ahora se ha elegido implementar el sistema utilizando el estilo *Client-Server* [GS96]; la agencia de noticias es el *Servidor* y las redacciones son los *Clientes*. El *Servidor* mantiene una lista de noticias pendientes para cada *Cliente* que no haya manifestado que no quiere recibir noticias. Cada vez que se genere una noticia, la misma se añadirá a todas las colas. Los *Clientes* consultan periódicamente al *Servidor* por nuevas noticias, si la cola para el *Cliente* que realice la consulta no está vacía el *Servidor* responderá que existe una nueva noticia y retirará la misma de la cola. De esta forma, la solución propuesta resiste las limitaciones impuestas por la existencia de los firewalls y las redacciones pueden estar al tanto de todas las noticias generadas por la agencia de noticias.

La forma en que las soluciones fueron descriptas y declaradas “correctas”, sufre de cierta liviandad (o levedad para los seguidores de Kundera) intencional en aras de presentar el tema y de dar una noción del contexto práctico del mismo. En la siguiente sección se analizara las implicaciones de esta liviandad y se explica cómo se lidia con la misma.

El objetivo de esta tesis es contestar una serie de preguntas acerca del uso de las descripciones arquitectónicas como herramientas prácticas para la comunicación, el diseño, el análisis y la construcción de un sistema. ¿Es realmente la primera descripción presentada insatisfactoria para el sistema? ¿Es la segunda realmente satisfactoria? ¿Existe una relación entre las dos descripciones? ¿Se pueden utilizar las dos descripciones en conjunto? ¿Una descripción provee más información que la otra? Para lograrlo se presentará y estudiará un tipo de relación que puede darse entre dos descripciones arquitectónicas de un mismo sistema a la que se llamará **refinamiento**. A tal fin, se proveerán las herramientas necesarias para el estudio formal de las descripciones arquitectónicas y sus propiedades.

1.2 Liviandad, folklore y software

La "liviandad" (en la forma de falta de formalidad) y las descripciones arquitectónicas han ido siempre de la mano, más allá de varios intentos por establecer descripciones formales mediante Lenguajes de Descripción Arquitectónica o ADLs por sus siglas en ingles [GS96]. Sin embargo, esto no fue obstáculo para que las descripciones arquitectónicas se hayan consolidado como elemento de comunicación y documentación en la práctica (principalmente en forma de diagramas).

¿Cómo se llega a la conclusión de que cualquiera de los diagramas arquitectónicos presentadas anteriormente (que no son más que cajas y líneas o "box+lines" que hacen las veces de una descripción arquitectónica informal e incompleta) son buenos o malos? ¿Cómo es que logran comunicar si no tienen una semántica asociada? La respuesta es que detrás de estos diagramas existe la "semántica suficiente" para cumplir los objetivos planteados.

Un diagrama presenta información muy básica de un sistema (por ejemplo, la existencia de ciertos nombres en el contexto del mismo) y generalmente viene acompañado de una breve descripción complementaria, en forma de comentarios escritos u orales. El receptor de este diagrama combina esta información con sus propios conocimientos previos, para terminar de formar un concepto del sistema en cuestión y así determina los roles y comportamientos de los diferentes elementos[2] del mismo. El conocimiento previo es el "folklore" de desarrollo: nombres y patrones comunes que comunican implícitamente las intenciones del autor del diagrama. Por ejemplo, si una "caja" de un diagrama lleva por nombre "Servidor de datos" y otra caja lleva por nombre "Cliente", entonces es plausible imaginar que se intenta comunicar la presencia de un patrón del estilo *Client-Server*. Aunque no exista una definición formal que dictamine que es el "estilo *Client-Server*" (por lo menos, no una universalmente aceptada) toda persona involucrada en el mundo del desarrollo de software tiene una idea generalizada de lo que es (y de lo que NO es). Es así que se obtiene una imagen mental del sistema, una metáfora del mismo [Holt01]. Lo sorprendente es que muchas veces la metáfora alcanzada por las diferentes personas que interactúan con un diagrama (realizándolo o leyéndolo) es, a muy grandes rasgos, la misma; es una metáfora compartida [AAG95].

Las "propiedades" exhibidas por las descripciones son aquellas que se pueden inferir de la metáfora compartida (parcial e informal) del sistema. Sí se dijo que una descripción es adecuada, es porque pudo determinarse a partir de la metáfora compartida. Una descripción arquitectónica puede pensarse entonces como una especificación: a mayor calidad (con calidad = formalidad = no ambigüedad) serán más y más complejas las propiedades que se puedan inferir. En el desarrollo de esta tesis se formalizarán varios de los conceptos presentados hasta aquí, en aras de poder estudiar las descripciones arquitectónicas con la suficiente profundidad para lograr los objetivos propuestos para el trabajo.

[2] Se utiliza la palabra "elementos" y no "componentes" o similar para mantener la idea expuesta libre de términos que sugieran segundas intenciones o conceptos más profundos.

1.3 Descripciones y sus roles

La primera descripción arquitectónica propuesta para el sistema de la agencia de noticias se determinó no satisfactoria, debido a un requerimiento no funcional que dictamina ciertas restricciones sobre el medio de comunicación a utilizarse entre las partes. Antes de presentar este requerimiento, la descripción se perfilaba adecuada por los siguientes motivos:

- Resuelve el "core" del problema propuesto eficientemente mediante el envío de eventos
- Exhibe alta cohesión, las responsabilidades están bien definidas
- Exhibe bajo acoplamiento, las redacciones y la agencia de noticia no se encuentran relacionadas directamente
- Los "design rationales" (las propiedades deseables y como éstas se logran) están claramente expuestos gracias al uso de un estilo "conocido en el folklore"

En definitiva, el diagrama era una buena descripción del sistema que no considera detalles que pueden tildarse de implementativos (la existencia de firewalls). Es una visión conceptual del sistema, más cercana al dominio propio del problema a resolver que a una potencial implementación. Sin embargo, esta cercanía al dominio (y lejanía de la implementación) no considera los conflictos que pueden surgir a la hora de construirlo [McConnell04]. Estos pueden ser:

- Falta de herramientas para implementar en forma directa la solución planteada
- Aun contando con las herramientas adecuadas, hay varios detalles prácticos que no se contemplan (por ejemplo, la existencia de firewalls)

El resultado final esperado para un proceso de desarrollo de software es una implementación funcional del sistema y para lograrlo este debe construirse. El problema es que la metáfora propuesta es buena, pero insuficiente para llevar cabo la tarea de construcción. La solución es desarrollar una nueva metáfora que exhiba las bondades de la anterior y que aporte nueva información que permita facilitar el proceso de construcción. En definitiva, se necesita **reconsiderar el nivel de abstracción** con el que se está trabajando para obtener una nueva metáfora más acorde a las necesidades de construcción (que exhiba los detalles necesarios para tal tarea). La nueva metáfora debería tener todas las bondades de la anterior y a la vez paliar sus dificultades.

La segunda descripción presentada soluciona el problema práctico de los firewalls al proveer mayor detalle sobre los aspectos de comunicación y al "emular" el comportamiento de envío de mensajes exhibida por la anterior (una buena propiedad de la primera descripción). Por otra parte, parecería exhibir un grado mayor de acoplamiento entre las partes y los "design rationales" ya no son tan claros (lo cual queda evidenciado por la necesidad de incluir comentarios textuales). La

segunda descripción es mejor para construir, pero la primera es mejor para comunicar el concepto global del sistema. Las dos representan un mismo sistema, pero lo exhiben a diferentes niveles de abstracción (al utilizar abstracciones diferentes).

Se puede decir que la segunda descripción es un **refinamiento** de la primera: una segunda metáfora que representa al mismo sistema que la primera, pero con un grado de detalle mayor. El grado de detalle mayor altera tanto las propiedades que se exhiben (tal vez algunas se pierdan, como la exhibición de cohesión), como la efectividad para exhibirlas. Un buen refinamiento permitiría mantener la mayoría de las propiedades "buenas" de la descripción refinada, las exhibiría con (casi) la misma efectividad y aportaría nuevas propiedades "buenas" que no podían ser exhibidas en la descripción original (por su nivel de abstracción).

El objetivo de esta tesis es contestar, a partir del estudio del concepto de refinamiento arquitectónico, las preguntas planteadas en la primera sección sobre el uso de las descripciones arquitectónicas como herramientas prácticas para la comunicación, el diseño, el análisis y la construcción de sistemas.

1.4 Organización del resto del trabajo

El resto del trabajo se organiza de la siguiente forma: En el capítulo siguiente se presentan y definen los conceptos de arquitecturas de software necesarios para comprender el desarrollo del trabajo. En el capítulo tres se desarrolla el concepto de refinamiento arquitectónico; se presenta un marco conceptual adecuado para su estudio y se proponen los fundamentos de una metodología de refinamiento arquitectónico a partir del uso de estilos. El capítulo cuatro utiliza los aportes del anterior para presentar una metodología de refinamiento arquitectónico utilizando el model checker LTSA; se incluyen varios ejemplos para ilustrar y validar la propuesta. Se finaliza presentando posibles líneas de trabajo futuro y las conclusiones a las que se arribo a partir de la elaboración del presente.

2. Conceptos básicos

La arquitectura de software, como campo de estudio, sufre de un problema fundamental y es que no existen definiciones universalmente aceptadas para varios de sus conceptos. Este problema resulta de dos motivos principales: Por un lado, la arquitectura de software tiene como particularidad que puede abordarse formal o informalmente, confiriendo una suerte de dualidad en su estudio; las definiciones utilizadas en los trabajos dependen directamente de la formalidad del mismo. Por otro lado, la relativa juventud del concepto (la gran mayoría de los trabajos fundacionales corresponden a la década del '90) impide la consolidación de definiciones por peso "histórico". Para evitar malos entendidos, en este capítulo se presentan las definiciones que serán adoptadas para el desarrollo de esta tesis.

2.1 Arquitectura de software

Lamentablemente no existe una única definición de arquitectura de software. De hecho, existen tantas definiciones diferentes que el Software Engineering Institue del Carnegie Mellon tiene una página web dedicada a recolectarlas todas [SEI]. En este trabajo se utiliza la siguiente definición, adaptada de [BCK03]:

> *La arquitectura de software de un sistema es la estructura o estructuras, que involucran a los diferentes elementos del software, sus propiedades externamente visibles y las relaciones entre ellos.*

La definición presentada está lejos de ser formal y da lugar a varios interrogantes, entre los que se pueden destacar:

- ¿Los principios y lineamientos que dictaminan la forma de las estructuras son parte de la arquitectura?
- ¿Se incluye la relación de las estructuras con su entorno como parte de la definición?
- ¿Las estructuras son de naturaleza estática o dinámica?

Todos estos interrogantes habilitan el estudio de diferentes aspectos del trabajo arquitectónico y las respuestas para los mismos dependen del autor y del contexto de aplicación. De esta forma, el estudio de las arquitecturas de software abarca un arco sumamente extenso de conceptos, que va desde aspectos puramente prácticos (como pueden ser diferentes formas de documentación arquitectónica) hasta aspectos más bien filosóficos (¿todo sistema tiene una arquitectura mas allá de que ésta no haya sido concebida por los desarrolladores?), pasando por aspectos puramente teóricos (Por ejemplo, model checking de descripciones arquitecturas).

En todo caso, lo laxo de la definición permite cierta libertad a la hora de trabajar con el concepto y resulta en la posibilidad de crear nuevas área de estudio. En cada uno de los siguientes trabajos

[BCK03, GS96, HNS99], se presenta un panorama amplio de las diferentes aplicaciones del concepto y de las ramas de estudio que éstas traen aparejadas.

Con respecto a los interrogantes planteados, en este trabajo se considera que todas las respuestas son afirmativas: Los principios y lineamientos son parte de la estructura, las estructuras incluyen la relación con su ambiente y son de naturaleza dinámica (aunque en este trabajo se estudiarán aspectos estáticos exclusivamente).

Por último cabe aclarar que, con el tiempo, las definiciones de arquitectura de software propuestas por diferentes autores han tendido a converger a un conjunto común de conceptos, por lo que se ha elegido una en particular para establecer claramente el marco de este trabajo y no porque una definición invalide necesariamente a las demás.

2.2 Vistas

Las descripciones arquitectónicas se han formalizado en la academia mediante el concepto de vistas arquitectónicas. Una vista arquitectónica es la representación de un conjunto de elementos arquitectónicos y sus relaciones [CBB+02]. En general, una vista representa elementos de una única estructura del sistema (ya que las relaciones entre elementos de diferentes estructuras son difíciles de establecer). Además múltiples vistas pueden representar una única estructura, de acuerdo a su complejidad.

Existen muchos trabajos que estudian las diferentes vistas que pueden manifestarse en el desarrollo de un sistema, sus beneficios y roles [BCK03, CBB+02, HNS99, Kru95]. En [CBB+02] se propone utilizar las siguientes vistas[3]:

- Vistas de módulos: Las vistas de módulos documentan las principales unidades de implementación de un sistema

- Vistas de componentes y conectores: Las vistas de componentes y conectores documentan las principales unidades de ejecución de un sistema. Estas unidades se clasifican en componentes (unidades de computo) y conectores (unidades de comunicación e interacción).

- Vistas de alocación: Las vistas de alocación documentan la relación entre el software y su entorno de ejecución y desarrollo

En general todos los trabajos sobre el tema incluyen las vistas anteriores o variaciones de las mismas, por lo que se puede considerar esta lista como un conjunto mínimo de vistas estándar. Se puede apreciar que cada una de las vistas enumeradas exhibe una estructura diferente del sistema. Este es el motivo por el cual todavía ningún trabajo pudo establecer un método práctico

[3] En [CBB+02] se distingue el concepto de vista y de tipo de vista (viewtype): Una vista pertenece a un viewtype determinado. Técnicamente hablando, la lista presentada es una lista de viewtypes y no de vistas. En casi todos los demás trabajos del área (incluyendo este) el término vista se utiliza para describir ambos conceptos.

para relacionar los diferentes tipos de vistas presentados; las estructuras son ortogonales entre si y abarcan aspectos heterogéneos del sistema.

Esta tesis propone trabajar exclusivamente con vistas de componentes y conectores de los sistemas. Sin embargo, se adoptara el nombre y la definición de [HNS99] para la misma. En [HNS99] la vista de componentes y conectores se llama vista conceptual y se establece que ésta debe describir al sistema en términos del dominio del mismo. Como resultado, no es necesaria la existencia de una relación simple entre la vista conceptual y la implementación de un sistema. Entonces, la vista conceptual permite representar el sistema mediante abstracciones complejas que no se encuentran necesariamente disponibles para la construcción del mismo. Dichas abstracciones son una representación fiel de la solución propuesta, mientras que la implementación se puede considerar accidental y acotada por las tecnologías existentes al momento del desarrollo. Llevando la idea a un extremo, se puede considerar a la vista conceptual como un espacio de construcción idealizado en el que se dispone de todas las abstracciones necesarias para expresar el dominio del problema y su solución. De esta forma se habilita un nuevo espacio de diseño que permitiría trabajar sobre dominios que involucren conceptos complejos y proveer soluciones acordes a los mismos.

2.3 Estilos

A través del estudio práctico de las descripciones arquitectónicas correspondientes a las vistas presentadas, diferentes autores han notado la manifestación de patrones recurrentes en los elementos y las relaciones exhibidas (aun en descripciones de sistemas heterogéneos) [AAG93, GS93, GS96]. Inspirados en el trabajo del arquitecto (arquitecto “tradicional”) Christopher Alexander [Alex77, Alex79] los patrones recurrentes se han formalizado mediante el concepto de estilo arquitectónico: Un estilo arquitectónico es un conjunto de reglas que define tipos de elementos y relaciones arquitectónicas y dictamina su uso válido. Si una vista arquitectónica respeta las reglas impuestas por un estilo, entonces se dice que la vista exhibe o hace uso de dicho estilo (y por transición, también lo hace la arquitectura representada).

Los estilos pueden determinar el comportamiento y otras propiedades de los elementos arquitectónicos y sus relaciones, pero no definen instancias específicas de los mismos [HNS99, BCK03]. De esta forma, los estilos definen implícitamente familias de arquitecturas a partir de la exhibición de las propiedades por estos definidos. El uso de un estilo por parte de una arquitectura, permite aplicar sobre la misma conocimiento y herramientas especializadas para dicho estilo [CBB+02]

Los estilos arquitectónicos cumplen varios roles en el marco de trabajo arquitectónico: En primer lugar, los estilos permiten capturar patrones de diseño adoptados por la comunidad y por lo tanto son un vehículo para representar el conocimiento comunitario en la materia. Los estilos facilitan el desarrollo de sistemas cada vez más complejos a partir del re-uso de conocimiento.

En segundo lugar, los estilos codifican el “design rationale” detrás de una arquitectura. Se considera que sí una arquitectura hace uso de un estilo determinado, entonces las reglas

aportadas por el mismo deberían reflejar (en cierta medida) las intenciones del diseñador: Por ejemplo, la exhibición del estilo *Announcer-Listener* en el ejemplo de la presentación permite suponer la intención de establecer un patrón de comunicación mediante mensajes, en el que el acoplamiento entre emisor y receptor sea mínimo.

Por último, un estilo se puede pensar como una gramática: define un lenguaje (mediante los tipos de elementos y relaciones definidos) y las reglas que dictaminan su uso. Al aplicar un estilo, se enriquece el vocabulario disponible para describir el sistema. Así se enriquece el vocabulario disponible para la comunicación entre los stakeholders del sistema. Volviendo al ejemplo de la introducción, la exhibición del estilo *Announcer-Listener* permite predicar sobre el sistema a partir de los términos (aportados por el estilo) "*Anouncer*", "*Listeners*" y "*Eventos*".

Dada la complejidad de algunos estilos, a veces estos se especializan en sub-estilos. Los sub-estilos particionan al conjunto de sistemas definidos por un estilo en sub-conjuntos, al imponer nuevas reglas de clasificación. En esta tesis los estilos y sub-estilos jugarán un rol fundamental al articular mecanismos de trazabilidad entre diferentes descripciones arquitectónicas de un mismo sistema.

2.4 Lenguajes de descripción arquitectónica

Un lenguaje de descripción arquitectónica (ADL por sus siglas en ingles) es un lenguaje (grafico, textual o hibrido) que se utiliza para describir un sistema a partir de sus elementos arquitectónicos y las relaciones entre estos [BCK03]. En este trabajo no se utiliza ningún ADL en particular, en aras de otorgarle la mayor generalidad posible a los resultados alcanzados. Las descripciones arquitectónicas se presentan mediante una combinación de la notación grafica utilizada en [HNS99] y código FSP explicito. De esta forma, se utiliza un enfoque similar al de [AAG95]. Esta elección no implica que los resultados obtenidos sean incompatibles con el uso de diferentes ADL's. En el capítulo "Trabajo Futuro" se puede encontrar más información al respecto.

3. Estudio del concepto de refinamiento arquitectónico

3.1 ¿Por qué es necesario refinar?

La idea de refinar descripciones arquitectónicas aparece junto a las primeras publicaciones del área de arquitecturas de software. Sin embargo, mientras que otros aspectos del trabajo arquitectónico han ido evolucionando hasta su consolidación definitiva (el concepto de vistas [CBB+02, HNS99, Kru95], el uso de estilos [AAG93, PW92], técnicas metodológicas de valoración y validación de arquitecturas [CKK01], etc.) el concepto de refinamiento ha sido relegado, generalmente, a menciones anecdóticas o resultados secundarios de algunos trabajos. Existen algunas pocas excepciones; trabajos como [Gar96] y [MQ94, MQR95] están dedicados exclusivamente al tema con resultados muy diferentes a los anteriores, pero la norma general ha sido una suerte de discriminación que puede ser el resultado de distintos factores:

- Falta de necesidad y/o aplicabilidad del concepto: Ocurre muchas veces que un concepto no surge en la comunidad académica y/o en la industria, porque el mismo no se considera necesario o aplicable; el concepto de refinamiento posiblemente sufra esta condición. La mayoría de los trabajos presentados hasta el momento son, en parte, responsables de esta realidad, ya que no suelen presentar un marco teórico-práctico que justifique la aplicación y estudio del concepto.

- La existencia de un concepto similar en áreas relacionadas: El concepto de refinamiento existe y es estudiado en el campo de las álgebras de procesos, máquinas de estados y sistemas de transición etiquetados (LTS) [Hoa85, Ros97]. Inclusive, existen herramientas que validan refinamientos en forma automática [FDR05]. Como resultado, no se cree necesario reformular un concepto ya existente.

- Falta de un marco formal para la aplicación del concepto: A diferencia de otras áreas de estudio en el desarrollo de software, el concepto de arquitecturas de software puede ser abordado desde puntos de vista no formales. De hecho, la mayoría de los trabajos publicados sobre el tema carecen de bases estrictamente formales. De las pocas bases teóricas formales presentadas, ninguna ha logrado imponerse en el ámbito académico y/o comercial por sobre las demás. La falta de un marco formal generalizado para el concepto de arquitectura bien pudo haber impedido el desarrollo del concepto de refinamiento, ya que él mismo necesita de herramientas con cierto grado de formalidad para su estudio extensivo y aplicación.

- Mal pandémico de la computación: el uso, sobre-uso y abuso de términos cuya definición no es del todo clara [Fowler03], a veces conduce a creer que el concepto ya está, de hecho, definido y adoptado por la comunidad (También conocido como el problema del “bastardeo”).

Con el permiso del lector y contagiándose brevemente del último punto citado, se postergará por unos párrafos la presentación de una definición satisfactoria del concepto de refinamiento en aras de lograr una mejor presentación del trabajo.

El concepto de refinamiento, con una definición adecuada para el marco de trabajo arquitectónico, puede resultar sumamente útil para la creación e implementación de abstracciones (o meta-abstracciones) que faciliten la construcción, documentación, análisis y comunicación de un sistema. Adoptando las nociones de [HNS99] se define el marco de trabajo arquitectónico a partir de los dos roles identificados para la arquitectura de un sistema: el de ser un plan de diseño (un plano del sistema en cuestión) y el de ser una abstracción idónea para la comunicación y el análisis. La habilidad de una descripción arquitectónica para cumplir con cualquiera de los dos roles está asociada directamente con el nivel de abstracción que ésta exhibe: solo es posible comunicar, analizar, planificar y construir lo que se expone y no lo que se oculta.

Retomando el ejemplo presentado en la Introducción de este trabajo, mientras que es plausible considerar el primer diagrama arquitectónico del ejemplo como una abstracción útil para la comunicación (ciertamente comunica la idea central del sistema y lo expone en términos adecuados al problema planteado) es por lo menos exagerado, presentarla como un "plano" útil para la construcción[4]. Haciendo uso de la analogía simplista, la descripción presentada es tan útil para la construcción de un sistema, como lo es un folleto publicitario inmobiliario para la construcción de un edificio (Mientras que poca gente se aventuraría a habitar una vivienda u oficina construida de tal forma, es asombrosa la cantidad de "profesionales" que consideran que la descripción arquitectónica presentada ES la arquitectura del sistema y que hasta allí llega su rol en el desarrollo del sistema, lanzándose así a la próxima etapa de la metodología en uso). La ineficacia del primer diagrama para ocupar el rol de plano de construcción resulta del nivel de abstracción elegido para su elaboración: la distancia entre las abstracciones utilizadas y las herramientas de implementación es tal, que impide la utilización efectiva de las mismas [Gar96]. Incluso contando con herramientas que puedan trabajar en el nivel de abstracción elegido (middleware especializado como, por ejemplo, Siena [Car98]), es muy difícil que éstas puedan aplicarse sin considerar un número determinado de detalles que no figuran en el diagrama.

El problema con el primer diagrama es el nivel de abstracción que exhibe. Éste debe ser reconsiderado, para que entre la información selectivamente ignorada [Str00] no se incluyan los detalles que la descripción anterior ocultaba deliberadamente. Es decir, se necesita una nueva descripción arquitectónica en la que, mediante un nivel de abstracción reconsiderado, se exhiban nuevos detalles de interés que en la primera descripción no se pueden encontrar. Para no perder los resultados del trabajo de diseño, la nueva descripción deberá mantener el "espíritu" de la original: las propiedades inferibles que sean de interés para el sistema, deben poder ser inferidas también a partir de la nueva descripción (Tal vez con un grado mayor de dificultad, debido al

[4] La noción de "plano" no debe asociarse exclusivamente con la vista de módulos [CBB+02]. En este trabajo se propone la utilización de la vista conceptual como "plano", ya que se considera la asignación de roles y comportamientos como parte fundamental del proceso de "construcción mental" del sistema.

nuevo nivel de abstracción adoptado). Se puede considerar, entonces, a la nueva descripción arquitectónica como un **refinamiento** de la primera:

Una reformulación que resulta de reconsiderar el nivel de abstracción, conservando el "espíritu" de la descripción original y tendiente a proveer un mayor grado de detalle para ciertos aspectos de interés.

El término "refinamiento" se utiliza de diferentes formas para referirse a:

1. La relación existente entre dos descripciones ("El concepto de refinamiento propiamente dicho")
2. El par de descripciones tales que existe una relación de refinamiento entre ellas ("La descripción A y la descripción B determinan un refinamiento válido")
3. La reformulación de la descripción original ("La descripción B es un refinamiento válido para la descripción A")

Del contexto se debería determinar cuál de estos usos se está aplicando.

¿Cuál es la ventaja de contar con el concepto de refinamiento? En primer lugar, la capacidad de transitar el gap conceptual, cognitivo y semántico que existe entre un diseño y su implementación con cierto nivel de confianza. Un gap diseño-implementación menor reduce la posibilidad de encontrar errores de "conformance" [MNS95] y de "mismatching" entre elementos arquitectónicos [GAO95] que impidan implementar la solución diseñada. También reduce las posibilidades de generar errores de implementación, ya que ésta estaría especificada en cierto grado por el diseño. Cada vez que una descripción arquitectónica es refinada, se obtiene una nueva descripción más cercana a la implementación final (conceptual, semántica y cognitivamente hablando), que conserva los fundamentos de las descripciones anteriores (y más lejanas) y describe nuevos aspectos del diseño a un nivel conveniente de abstracción. Existe entre una descripción y su refinamiento, una suerte de trazabilidad (basada en los fundamentos conservados) que permite relacionarlas.

Como todo gran viaje, la construcción de un sistema comienza con un primer paso (lejano y abstracto) y debe realizarse paso a paso (para no perder los aportes de los pasos anteriores). Un proceso de refinamiento permite crear una jerarquía de descripciones en la que el nivel de abstracción va variando convenientemente para poder exhibir adecuadamente diferentes "propiedades de interés". El tamaño del gap entre la implementación final y cualquiera de las descripciones, es inversamente proporcional a la altura de la descripción en cuestión dentro de la jerarquía. Volviendo al ejemplo de la introducción, el primer diagrama transmite un concepto de muy alto nivel del sistema: la comunicación entre las partes debe seguir un patrón del tipo *Announcer-Listener*. No comunica como este patrón será implementado, ni hace referencia a ninguna solución de construcción. El segundo diagrama, por su parte, efectivamente comunica

una solución de construcción[5] (o, por lo menos, una solución más cercana a la realidad) que respeta el patrón presentado en la descripción anterior. El segundo diagrama se encuentra entonces, más cercano a la implementación final y por lo tanto el gap entre éstos es menor al existente entre el primer diagrama y la implementación. Con cada refinamiento posterior, el gap se reducirá todavía más y de esta forma se transita el mismo con la confianza que resulta de contar con trazabilidad entre las descripciones.

Otra ventaja de contar con el concepto de refinamiento es que al refinar, se van ganando diseños más detallados (y correctos con respecto a los diseños anteriores al conservar sus "fundamentos"), que pueden ser útiles para la validación temprana de la capacidad del sistema para satisfacer requerimientos funcionales y no funcionales. Existen numerosos tipos de análisis que pueden realizarse sobre los diseños [SW99] que permiten inferir esta información. Cada uno de estos análisis requerirá que la arquitectura se encuentre descripta mediante un nivel de abstracción apropiado.

En los párrafos anteriores se mencionan las "propiedades interesantes" de una descripción arquitectónica. Éstas no son otra cosa más que la materialización en el diseño, de la satisfacción de los requerimientos (funcionales y no funcionales) del sistema. El objetivo del proceso de diseño es que el modelo exhiba estas propiedades. Al intentar satisfacer este objetivo, se aplican diferentes **design rationales** en la elaboración del modelo del sistema. Los design rationales aplicados desembocan (o condicionan) en estrategias para la implementación. Dichas estrategias obligarán a validar y, eventualmente, repensar los requerimientos iníciales. La capacidad de codificar los principios por los cuales se pueden justificar la implementación del sistema, es la causa por la cual varios autores [HNS99] argumentan que la arquitectura de un sistema es una herramienta ideal para presentar el sistema a nuevos miembros del equipo de desarrollo del mismo o, como se puede leer en [Naur85], transmitir la **teoría** del sistema. La siguiente figura ilustra la interacción design rationales – requerimientos – implementación – teoría:

[5] Ya que el patrón *Client-Server* es mucho más cercano a las abstracciones computacionales generalmente disponibles para construir software

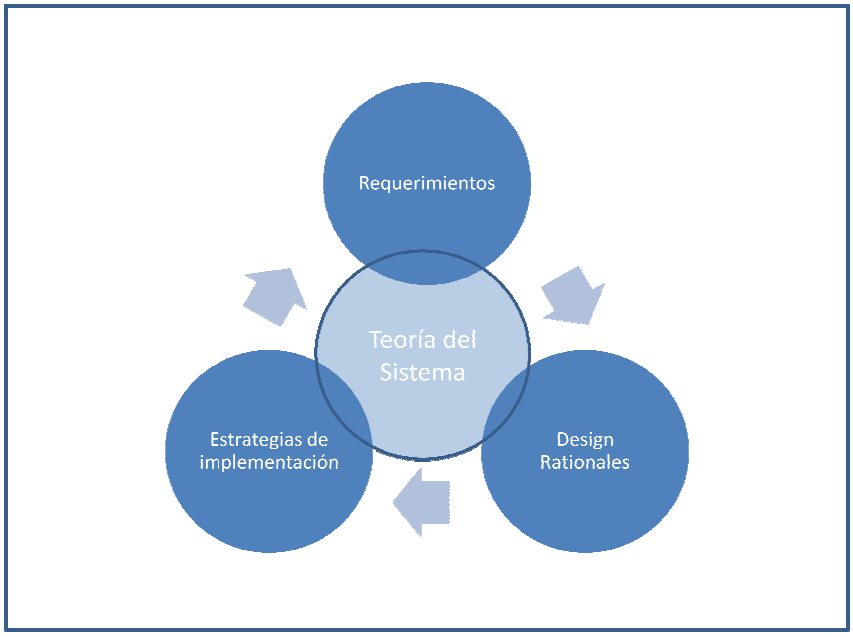

Figura 3-1: Interacción design rationales – requerimientos – implementación – teoría

En [BRJ99] se afirma que "Todo modelo puede ser expresado a diferentes niveles de detalle" y que la existencia de estos niveles de "detalle" (o abstracción en el lenguaje utilizado en este trabajo) permite "digerir" más fácilmente la complejidad del ente modelado. Siendo la arquitectura de un sistema un modelo del mismo y siendo el software complejo por naturaleza [Brooks87], entonces es lógico pensar que sea beneficioso presentar su teoría mediante diferentes grados de abstracción. Esto es, precisamente, lo que provee una jerarquía de refinamientos y es otra ventaja de contar con este concepto.

Ahora, si la descripción resultante de un proceso iterativo de refinamiento (la descripción 3, refina a la descripción 2, que refina a la 1) incluye las "propiedades interesantes" de las descripciones anteriores… ¿Quiere esto decir que dicha descripción puede reemplazar a las anteriores? ¿Las descripciones anteriores son desechables? La respuesta es NO y la justificación más básica para la misma se podría basar en un principio básico de la Ingeniería de Software, mientras más documentación exista (actualizada y correcta, por supuesto) mejor[6]. Sin embargo, la respuesta es NO debido a que las descripciones anteriores son mucho más adecuadas para exhibir las "propiedades interesantes" que introdujeron al diseño. Al eliminar las descripciones anteriores, se elimina implícitamente toda prueba que justifica por qué la descripción arquitectónica menos abstracta llego a ser lo que es. Retomando el ejemplo del primer capítulo, la segunda descripción puede exhibir la forma en la que los componentes interactúan para lograr la funcionalidad correcta del sistema, pero solo mediante un análisis más profundo (¡Análisis innecesario ya que fue realizado con anterioridad!) se puede determinar que se está ante un clásico esquema de

[6] Lo cual no se contradice con los principios expuestos por las metodologías ágiles. Estas evitan crear documentación por considerarlo superfluo o entorpecedor, pero no se oponen a la existencia de documentos mientras que no perjudiquen la tarea de implementación propiamente dicha

anuncio de eventos. Es decir, las descripciones anteriores proveen el contexto necesario para interpretar la descripción en cuestión. La falta del contexto adecuado, no permite comprender el sistema en toda su dimensión. Se analiza el siguiente escenario a modo de ejemplo: Cuando un programador debe trabajar con código escrito por un segundo programador, se puede dar una de las siguientes situaciones:

1. El programador no comprende el código, desprecia al segundo programador y todo cambio en el código se realiza en forma de parches

2. El programador comprende el código, pero no entiende porque el sistema fue programado de esa manera. Desprecia al segundo programador y todo cambio en el código se realiza "imitando" la forma del código actual

3. El programador comprende el código y los motivos por los cuales el sistema fue implementado de esa manera (pero desprecia al segundo programador de todas maneras). Los cambios "encajan armoniosamente" en el resto del código.

Dado la improbable de la situación número tres, se analizan las otras dos: En ambos casos la situación mejoraría notablemente, si el programador original pudiera explicarle al nuevo los motivos por los cuales llego a determinar que el código escrito es el mejor posible para el sistema. Seguramente, el programador original justificaría su código a partir de diferentes restricciones que existieron al momento de escribirlo (tiempo, herramientas disponibles, voluntad, etc.) y a partir de cierto proceso mental (que incluye su concepción mental del sistema) que él mismo realizó previo a la escritura del código. En definitiva, el programador original podría decirle al nuevo "Este código, pensalo de esta manera… miralo de este modo… considerá todas estas cosas…". Este es el tipo de información que podría ser aportada por una jerarquía completa de refinamientos; información que completa la visión que el nuevo programador puede tener sobre el sistema (Aunque seguramente no disminuirá el desprecio natural que sienten los programadores entre sí). De esta manera, la situación número tres podría ser mucho más común de lo que es hoy en día. En resumen, una jerarquía de refinamientos completa permite comunicar la teoría del sistema eficientemente.

Por otra parte, la jerarquía completa de refinamientos es una suerte de historial de diseño, un "log" del trabajo intelectual de diseño. Al igual que una base de datos, que mediante su log puede ser revertida ("rolled back") a un punto dado en su historial, si en algún momento se toma una mala decisión de diseño (una que desemboca en una mala implementación) se puede volver el "tiempo atrás" hasta el último diseño correcto en la jerarquía.

El concepto de refinamiento puede ser utilizado también, como una herramienta extra en un proceso de ingeniería inversa. Hasta aquí se presentó el concepto de refinamiento como una herramienta para ser aplicada mediante una estrategia top-down, pero también puede aplicarse en el marco de una estrategia bottom-up para crear abstracciones útiles a partir de un sistema existente. De esta forma, se podría tratar de regenerar la "teoría perdida" del sistema. El concepto

de refinamiento se podría aplicar para potenciar el uso de herramientas de descubrimiento de arquitecturas como DiscoTect [YGS+04].

Todas las razones vertidas por los párrafos anteriores son suficientes para justificar el estudio del refinamiento como una herramienta útil para la construcción iterativa e incremental de sistemas. Así se hace frente a la primera causa de "discriminación" presentada al comienzo de la sección (Falta de necesidad y/o aplicabilidad del concepto). En la siguiente sección se hará frente a la segunda causa (Existencia de un concepto similar en otras áreas) al justificar por qué las nociones clásicas de refinamiento son insuficientes e inadecuadas para el marco de trabajo arquitectónico.

3.2 Nociones anteriores de refinamiento

La noción clásica de refinamiento surge del área de trabajo del álgebra de procesos, máquinas de estados y sistemas de transición etiquetados (Labelled Transtition Systems, LTS). Si bien en este área existen diferentes definiciones y tipos de refinamientos [Ros97], todos están basados en la noción de behavior substitutability: Una entidad (Proceso, FSM o LTS) refina a otra, sí la primera puede sustituir a la segunda sin que un agente externo al concepto pueda notar diferencia alguna. De esta forma, dos descripciones arquitectónicas serían iguales si el refinamiento no produce ningún comportamiento externamente visible que la descripción original no pudiera haber producido. La definición presentada es útil solo para un único escenario de refinamiento arquitectónico: cuando el mismo es composicional y preserva las estructuras originales. En este tipo de refinamiento, cada uno de los elementos que componen la descripción original "explota" en un número dado de nuevos elementos en forma estrictamente jerárquica. La nueva descripción se puede particionar en subconjuntos disjuntos, donde cada uno de ellos refina/sustituye a un único elemento de la descripción original.

Lamentablemente, como fuera adelantado en el párrafo anterior, la noción clásica de refinamiento no es adecuada para el trabajo arquitectónico: El problema es que la definición basada en behavior substitubility no toma en consideración las características únicas del trabajo arquitectónico. La noción clásica, en cualquiera de sus definiciones, se basa en la capacidad de sustituir un concepto por otro, pero en el marco arquitectónico un refinamiento debe complementar al concepto original y no sustituirlo.

Un modelo arquitectónico es una descripción parcial de un sistema con cierto nivel de abstracción. El nivel de abstracción se elige para que la descripción arquitectónica exhiba clara y explícitamente ciertas propiedades de interés. A menos que se tenga la habilidad de generar modelos perfectos, junto a éstas propiedades de interés también se exhibirán ciertas propiedades "accidentales" que resultan de las herramientas utilizadas para construir el modelo y de su naturaleza parcial. De acuerdo a la noción basada en behavior substitubility un refinamiento debería exhibir el mismo comportamiento, caracterizado tanto por las propiedades de interés como por las propiedades accidentales. Claramente, éstas últimas no son parte del "espíritu" de la descripción inicial y no tienen por qué conservarse en un refinamiento (de hecho las palabras "espíritu" y "accidental" raras veces deben encontrarse en una misma frase). Además, una descripción arquitectónica se puede utilizar para exhibir propiedades más allá del comportamiento externamente visible, como

puede ser la distribución de roles en la solución y la imposición de una estructura que determine la comunicación permitida entre los diferentes componentes del modelo (configuración [CBB+02]).

Moriconi y Qian [MQ94, MQR95] presentan uno de los primeros trabajos en los que se propone una definición alternativa de refinamiento, específica para el trabajo arquitectónico. Los autores reconocen que es necesario contar con una definición de refinamiento propia para el trabajo con arquitecturas, que garantice la conservación lo que denominan la *rason d'être* de la descripción arquitectónica original (lo que en este trabajo se denomina "espíritu", pero con sofisticación mediterránea). A tal fin, proponen que las descripciones arquitectónicas deben asumirse completas: las propiedades que el sistema debe exhibir son **únicamente** aquellas que puedan ser inferidas de la descripción; si una propiedad no puede ser inferida, entonces el sistema NO debe exhibirla. Por lo tanto, un refinamiento no podrá introducir nuevas propiedades que sean contrarias a las exhibidas por la descripción original porque TODAS las propiedades de ésta deben ser exhibidas por el sistema. El trabajo plantea que descripciones arquitectónicas y estilos arquitectónicos pueden ser considerados teorías de primer orden. La teoría correspondiente a una arquitectura incluirá entre sus axiomas todos los axiomas de los estilos que utilice. Para que una descripción arquitectónica refine a otra, deberá encontrarse un mapping "de interpretación" entre las formulas de las teorías de éstas. Dadas las teorías Θ y Θ', I es una mapping de interpretación si para toda sentencia F:

$$F \in \Theta \Rightarrow I(F) \in \Theta' \ \wedge \ F \notin \Theta \Rightarrow I(F) \notin \Theta'$$

La propiedad expuesta se conoce como *fidelidad* o *faithfulness* y es necesaria para que el mapping haga honor a la noción de completitud propuesta en el trabajo. Se dice entonces que el refinamiento es correcto con respecto a la interpretación encontrada. El mapping de interpretación es divido en dos partes: un mapping de nombres y un mapping de estilos. Mientras que el primero es especifico para cada refinamiento, el segundo puede generalizarse para crear patrones de refinamiento cuya validez es necesaria probar una única vez. Se propone, luego, utilizar los patrones para crear jerarquías de refinamientos en forma incremental.

La propuesta de Moriconi y Qian es acertada en tres aspectos: la presentación de una definición de refinamiento específica para el trabajo arquitectónico, la inclusión en la definición de una noción de validez relativa a una interpretación específica y la propuesta de utilizar patrones de refinamiento basados en estilos. Sin embargo, toda la propuesta es herida de muerte por la asunción de completitud utilizada. Ésta es artificial y va en contra de la naturaleza parcial de las descripciones arquitectónicas. Mediante la noción de completitud, se pretende que las descripciones estén libres de "propiedades accidentales". El resultado es una definición de refinamiento demasiado fuerte para el marco de trabajo pretendido, que afecta fuertemente la aplicabilidad de la metodología propuesta.

Garlan en [Gar96] aporta una nueva visión al concepto de refinamiento arquitectónico: Llevando la idea de Moriconi y Qian de contar con patrones de refinamiento un paso más adelante, propone el concepto de refinamiento de estilos arquitectónicos. Mientras que Moriconi estudia el refinamiento de sistemas particulares, Garlan propone establecer la existencia de una relación de

refinamiento entre estilos: si se determina que el estilo S_2 puede refinar al estilo S_1, entonces **cualquier** descripción arquitectónica expresada mediante S_1 puede ser refinada en una segunda descripción expresada mediante S_2. La cardinalidad existente entre estilos y sistemas determina el salto cualitativo que el concepto plantea. Reconociendo que un estilo define un conjunto demasiado heterogéneo de sistemas, Garlan replantea su idea y propone refinar en base a sub-estilos. Resulta de este planteo, una metodología basada en dos funciones: La primera determina un sub-estilo adecuado para los estilos exhibidos por el sistema a refinar. La segunda es una función de abstracción del estilo concreto (al que se refinará), a los sub-estilos identificados por la función anterior. En forma ortogonal a las ideas recientemente presentadas, Garlan concuerda con identificar al concepto de refinamiento con la idea de sustitución, pero hace notar que las definiciones anteriores de refinamiento han sido muy fuertes (Por ejemplo el planteo de Moriconi, por su noción de completitud) o inadecuadas (Por ejemplo cualquier noción basada en behavior sustitubility, no abarca aspectos ajenos al comportamiento) para ser útiles en la práctica. Propone entonces que la validez de un refinamiento debe ser formulada de la siguiente manera: Una descripción es un refinamiento válido si puede sustituir a la descripción refinada con respecto a un *conjunto acotado y conscientemente elegido de propiedades de interés*.

En el plano teórico la propuesta de Garlan es interesante, pero carece de lineamientos precisos que dicten cómo ésta debe ser llevada a la práctica. Es por esto que, tal vez, algunos problemas potenciales han pasado desapercibidos. La idea de trabajar con sub-estilos (y no con estilos propiamente dichos) puede ser llevada al extremo, argumentando que se debería trabajar con "sub-sub-estilos" y así crear una progresión hacia el infinito. En definitiva, es posible creer que se llegará a un punto en donde se identifiquen tanto sub-estilos diferentes, como sistemas que utilizan el estilo existan. De esta forma, las ventajas vaticinadas por el autor desaparecerían. Sí se toma la definición de estilo como un "template" que debe ser instanciado, es difícil ver como se puede predicar sobre todas las instancias sin caer en restricciones que vayan en contra del concepto mismo (aun considerando el uso de sub-estilos como unidad mínima de templating). Por último, el trabajo no explica cómo debería aplicarse la metodología sobre sistemas descriptos mediante estilos múltiples (Cualquier sistema con un grado mínimo de complejidad, se encuentra en este grupo).

Más allá de las críticas, [Gar96] realiza dos aportes fundamentales al estudio del concepto de refinamiento arquitectónico. El primero, es la identificación de los estilos como herramientas validas y sumamente importantes para lograr la consolidación del mismo. El segundo es el reconocimiento de la necesidad de contar con una definición del concepto que sea relativa a los intereses del ejercicio de diseño en el cual se desarrolla el refinamiento (Las *propiedades de interés*).

Al remarcar los problemas inherentes a cada una de las definiciones presentadas, se hace frente a la segunda causa de discriminación y se demuestra la necesidad de crear una nueva definición adecuada a los fines de esta tesis. En la siguiente sección se hace frente a la tercera de las causas de "discriminación" enumeradas en la sección anterior, al delinear un marco de estudio teórico-formal para el concepto de refinamiento.

3.3 Marco teórico-formal para el concepto de refinamiento

En esta sección se definen una serie de conceptos necesarios para la obtención de un marco teórico-formal para el estudio del concepto de refinamiento arquitectónico. De esta forma, se intenta hacer frente a la tercera de las causas de "discriminación" enumerada al principio de este capítulo. En el marco propuesto se utilizan, exclusivamente, descripciones arquitectónicas de la vista conceptual. Esto obedece a dos motivos principales: El primero es que de todas las vistas "estándar" [CBB+02, Kru95, HNS99], la conceptual es la menos atada a conceptos computacionales clásicos. Entonces, se posiciona, como la mejor vista para expresar abstracciones propias del dominio y la teoría de un sistema [HNS99]. En segundo lugar, el resto de las vistas "estándar" representan estructuras que se corresponden con elementos implementativos y/o físicos; por ejemplo, la vista de módulos representa las unidades de implementación y la vista de alocación representa a los entornos de ejecución y al hardware. Los elementos físicos (el hardware) se manifiestan en el espacio y cualquier representación de los mismos puede ser asociada a esta manifestación. De esta forma se crea un marco de referencia único para estudiar las relaciones existentes entre las diferentes descripciones, a partir de la relación entre éstas y los entes físicos representados. Lo mismo ocurre con las unidades de implementación, aun siendo intangibles: Los módulos se manifiestan mediante cierto código fuente, lo mismo ocurre con las clases, las funciones y cualquier otra herramienta implementativa mínima de los lenguajes modernos[7]. Las variaciones en los niveles de abstracción de las diferentes descripciones se encuentran restringidas por este "attach" a los entes manifestables representados. Por el contrario, las vistas conceptuales representan abstracciones propias del dominio de un problema. Al no existir necesariamente un marco de referencia único, los cambios en los niveles de abstracción pueden resultar en relaciones complejas y dignas de ser estudiadas.

A continuación se presentan los conceptos que conforman el marco propuesto.

Descripciones Arquitectónicas

Sean $Rols$ el conjunto universal de todos los roles, $Ports$ el conjunto universal de todos los puertos, $Components$ el conjunto universal de todos los componentes y $Connectors$ el conjunto universal de todos los conectores, una **descripción arquitectónica** se formaliza como una siete-upla de la forma:

$$\boldsymbol{AD} \equiv < CN, CX, P, R, \overset{\boldsymbol{R}}{\rightarrow}, \overset{\boldsymbol{P}}{\rightarrow}, \overset{\boldsymbol{AD}}{\leftrightarrow} >$$

con

1. $CN \subseteq Components,\ CX \subseteq Connectors,\ P \subseteq Ports,\ R \subseteq Rols,$
2. $\overset{R}{\rightarrow} : Roles \rightarrow Connectors,\ Dom\left(\overset{R}{\rightarrow}\right) = R \ \wedge\ Img\left(\overset{R}{\rightarrow}\right) = CX$

[7] Los lenguajes orientados a objetos puros tienden a minimizar este fenómeno, pero no pueden librarse del mismo totalmente

3. $\overset{P}{\rightarrow} : Ports \rightarrow Components,\ Dom\left(\overset{P}{\rightarrow}\right) = P \ \wedge Img\left(\overset{P}{\rightarrow}\right) = CN$

4. $\overset{AD}{\leftrightarrow} : ConnectionPoints \times ConnectionPoints$

5. $\overset{AD}{\leftrightarrow}\ simetrica\ \wedge e_1 \overset{AD}{\leftrightarrow} e_2 \Rightarrow ((e_1 \in R\ \wedge\ e_2 \in\ P) \vee\ (e_1 \in P\ \wedge\ e_2 \in\ R))$

6. $(\forall r{:}\, R\ \exists p{:}\, P) r \overset{AD}{\leftrightarrow} p,\ (\forall p{:}\, P\ \exists!\, r{:}\, R)\ r \overset{AD}{\leftrightarrow} p$

donde:

1. $ConnectionPoints = Rols\ \cup Ports$

Es decir, una descripción arquitectónica es un conjunto de conectores y componentes interconectados entre sí. Cada componente tiene uno o más puertos asociados y cada conector tiene uno o más roles asociados. Cada rol se encuentra ligado a uno más puertos y cada puerto se encuentra ligado a un único rol. La relación $\overset{AD}{\leftrightarrow}$ se extiende al conjunto $ArqElements$ de la siguiente forma:

$$\overset{AD}{\leftrightarrow} : ArqElements \times ArqElements, \qquad (r{:}\, R, p{:}\, P)\ r \overset{AD}{\leftrightarrow}\ p\ \Rightarrow \overset{R}{\rightarrow}(r) \overset{AD}{\leftrightarrow}\ \overset{P}{\rightarrow}(p)$$

donde:

1. $ArqElements = Components\ \cup Connectors$

Dado que definir explícitamente cada uno de los siete elementos que definen una descripción arquitectónica puede resultar tedioso, se apela al uso de diagramas como los utilizados en la introducción de esta tesis para su definición. Por ejemplo, el diagrama que se presenta a continuación:

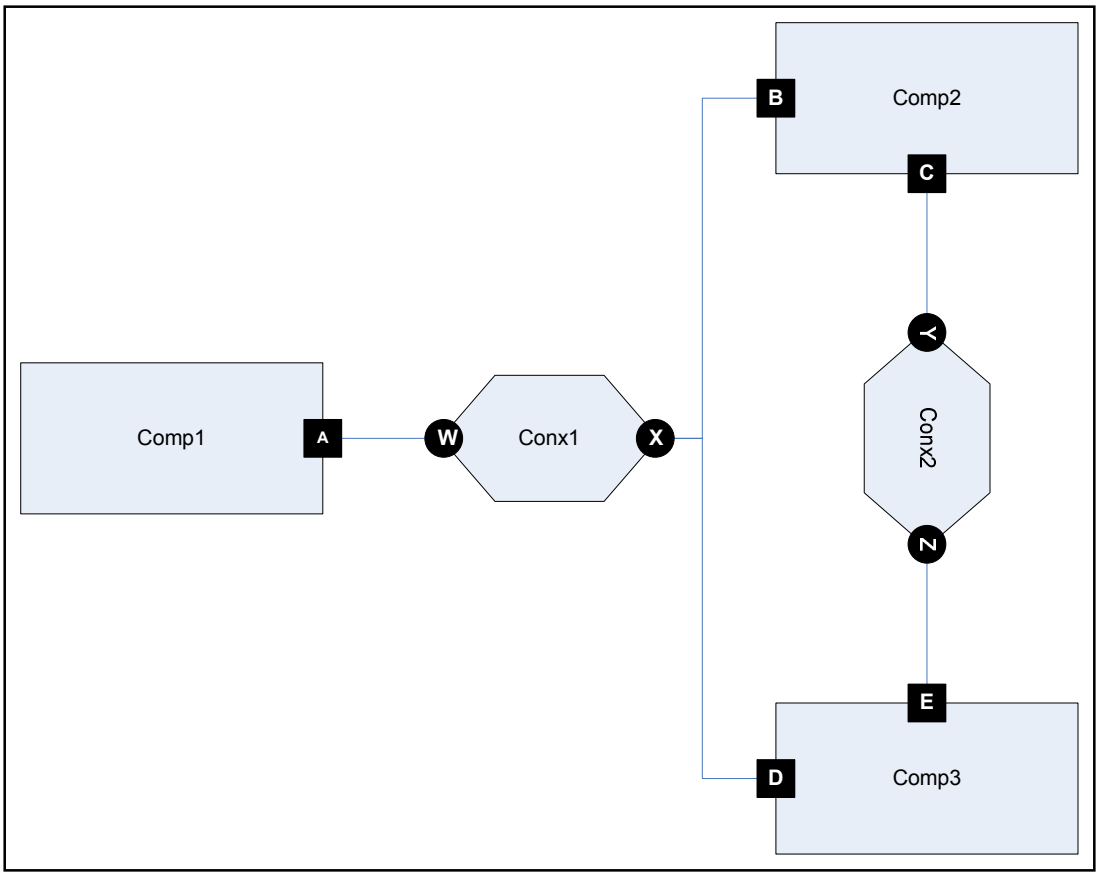

Figura 3-2: Diagrama arquitectónico de ejemplo

es equivalente a la descripción arquitectónica $< CN, CX, P, R, \overset{R}{\rightarrow}, \overset{P}{\rightarrow}, \overset{AD}{\leftrightarrow} >$ con:

- $CN = \{Comp1, Comp2, Comp3\}$
- $CX = \{Conx1, Conx2\}$
- $P = \{A, B, C, D, E\}$
- $R = \{W, X, Y, Z\}$
- $A \overset{P}{\rightarrow} Comp1,\ B \overset{P}{\rightarrow} Comp2,\ C \overset{P}{\rightarrow} Comp2,\ D \overset{P}{\rightarrow} Comp3,\ E \overset{P}{\rightarrow} Comp3$
- $W \overset{R}{\rightarrow} Conx1,\ X \overset{R}{\rightarrow} Conx1,\ Y \rightarrow Conx2,\ Z \overset{R}{\rightarrow} Conx2$
- $A \overset{AD}{\leftrightarrow} W,\ B \overset{AD}{\leftrightarrow} X,\ D \overset{AD}{\leftrightarrow} X,\ C \overset{AD}{\leftrightarrow} Y,\ Z \overset{AD}{\leftrightarrow} E$

La notación grafica se basa en la utilizada en [HNS99]. En ésta, los rectángulos representan componentes, los hexágonos estirados representan conectores, los círculos representan roles (asociados al conector representado por el hexágono sobre los que se ubican) y los cuadrados

representan puertos (asociados al componente representado por el rectángulo sobre los que se ubican). Las líneas representan asociaciones entre puertos y roles.

A lo largo de este trabajo, los roles y los puertos no se nombrarán explícitamente ya que su existencia no aporta a la metodología desarrollada. Por lo tanto, se trabajará con la versión extendida a componentes y conectores de la relación $\overset{AD}{\leftrightarrow}$. Los roles y puertos se incluyen en la definición para que la misma sea consistente con la noción de descripción arquitectónica utilizada en los principales trabajos del área [CBB+02].

Una descripción arquitectónica solo especifica su comportamiento a medias. Los nombres de los conectores y de los componentes dan una idea ad-hoc del comportamiento esperado (a partir del conocimiento comunitario en la materia y del uso de convenciones), pero no se define formalmente el comportamiento individual de cada uno de los elementos de de la arquitectura ni el resultado global de su interacción. Muchas veces este nivel de detalle es suficiente para comunicar y analizar una arquitectura a alto nivel, pero no es el caso para la aplicación de la metodología propuesta. En [HNS99] se propone interpretar las descripciones de la vista conceptual de la siguiente forma: cada componente y conector se considera un proceso que se ejecuta en forma independiente de los demás, el comportamiento de la arquitectura es el resultado de la ejecución simultánea de estos procesos. Siguiendo esta interpretación, es que se presenta a continuación el concepto de modelo de comportamiento y de comportamiento asociado a una descripción.

Modelos de comportamiento

Sea $ProcessesDescriptions$ el conjunto universal de las descripciones de procesos y $Properties$ el conjunto universal de las propiedades enunciables sobre el comportamiento de componentes, conectores e interacciones de las descripciones arquitectónicas, un **modelo de comportamiento** (o comportamiento a secas) es una tupla de la forma

$$\boldsymbol{B} \equiv < Procs, VerificableProps, BehaviorProps >$$

con:

1. $Procs \subseteq ProcessesDescriptions$
2. $VerificableProps \subseteq Properties$
3. $BehaviorProps \subseteq VerificableProps$

Un modelo de comportamiento es un conjunto de procesos junto a la especificación de, por un lado, las propiedades verificables sobre dicho conjunto (en la forma del conjunto $VerificableProps$) y por otro, las propiedades que son efectivamente exhibidas por el mismo (en la forma del conjunto $BehaviorProps$).

Modelo de comportamiento asociado a una descripción arquitectónica

Sea $AD = < CN, CX, P, R, \overset{R}{\rightarrow}, \overset{P}{\rightarrow}, \overset{AD}{\leftrightarrow} >$ una descripción arquitectónica y $B =< Procs, VerificableProps, BehaviorProps >$ un modelo de comportamiento se dice que B es un **comportamiento asociado** a AD si existe una función

$$\psi: (Components \cup Connectors) \rightarrow ProcessDescriptions$$

tal que:

1. $Dom(\psi) = CN \cup CX$
2. $Img(\psi) = Procs$
3. ψ es biyectiva

Es decir un comportamiento asociado a una descripción arquitectónica es un modelo de comportamiento por el cual se puede asociar un único proceso a cada uno de los componentes y conectores de la descripción. A partir de aquí, cuando se hable de una descripción arquitectónica implícitamente se estará hablando de una descripción con un comportamiento asociado.

Propiedades de una descripción arquitectónica

En la sección inicial de este capítulo se presenta una definición informal de refinamiento que dicta que la descripción arquitectónica C es un refinamiento de la descripción arquitectónica A, si C es una reformulación de A que resulta de reconsiderar el nivel de abstracción para proveer un mayor grado de detalle para ciertos aspectos de interés y que **conserva el "espíritu"** de la descripción original. El "espíritu" citado en la definición se corresponde con el conjunto de "**Propiedades interesantes**" de una descripción arquitectónica. Las descripciones arquitectónicas son creadas para exhibir, comunicar y/o analizar propiedades del sistema modelado. Las "Propiedades interesantes" de una descripción arquitectónicas son aquellas que justifican la creación de la misma, al representar propiedades del sistema modelado que es de interés exhibir, comunicar y/o analizar.

Relación de refinamiento naive

A partir de la definición informal de refinamiento citada anteriormente, se puede aventurar una primera definición formal para el concepto de refinamiento: Sean A y C dos descripciones arquitectónicas, $B_A =< P_A, VP_A, BP_A >$ y $B_C =< P_C, VP_C, BP_C >$ comportamientos asociados a las mismas y PI_A el conjunto de propiedades interesantes de A, entonces:

$$PI_A \subseteq BP_A \subseteq VP_A$$

Todas las propiedades interesantes de A deben ser ciertas para el comportamiento asociado propuesto (y por lo tanto, también deben ser verificables). Si B_C es un buen refinamiento de B_A entonces la siguiente expresión debería ser verdadera:

$$PI_A \subseteq BP_C \subseteq VP_C$$

Sin embargo, no se puede garantizar que todas las propiedades interesantes en PI_A sean verificables ($PI_A \subseteq VP_C$) para el comportamiento B_C ya que este comportamiento define procesos **diferentes** asociados a componentes y conectores **diferentes** a los de A. Se puede decir que el **lenguaje** planteado por B_C es **diferente** al planteado por B_A; las propiedades del conjunto PI_A se encuentran expresadas en el lenguaje definido por B_A.

Lenguaje de una descripción

El comportamiento asociado a una descripción arquitectónica describe el comportamiento de la misma valiéndose de un "lenguaje" determinado. A éste lenguaje se lo llamará **lenguaje exhibido por un comportamiento asociado a una descripción arquitectónica** (Aunque, para hacerle un bien a la ecología, se puede acortar a **lenguaje de una descripción** cuando su pueda determinar del contexto cual es el comportamiento asociado). La naturaleza del lenguaje quedará determinada por las herramientas utilizadas para definir los procesos. Por ejemplo, si los procesos se definen en base a un álgebra de procesos como FSP, entonces el lenguaje consistirá en los eventos utilizados para la definición de los procesos. Si los procesos se definen mediante un formalismo orientado a objetos, el lenguaje consistirá en los objetos utilizados para la definición y en los mensajes que ellos se envían entre sí.

La definición del conjunto de propiedades verificables para un comportamiento asociado a una descripción debería estar relacionada directamente con el lenguaje que éste exhibe. Si el lenguaje incluye conceptos como *noticias* y *redacciones* entonces es plausible creer que propiedades como "las *noticias* llegan en orden a las *redacciones"* puedan ser verificadas[8] ante la descripción, mientras que propiedades totalmente ajenas como "dos cabezazos seguidos en el área es gol seguro" no lo sean (más allá de que es vox populi).

Relación de refinamiento con traducción

La relación de refinamiento se puede reformular para evitar el problema resultante por la diferencia de lenguajes. La solución es proveer un mecanismo de traducción que permita expresar B_C en un lenguaje adecuado, de forma tal que las propiedades del conjunto PI_A puedan ser validadas sobre una traducción conveniente de B_C. A tal fin, se define el concepto de mecanismo de traducción. Un **mecanismo de traducción** es una función:

$$T: < Procs, VerificableProps, BehaviorProps > \rightarrow \\ < Procs, VerificableProps, BehaviorProps >$$

[8] Suponiendo que "orden" y "llegan" estén también en el lenguaje o su significado se entienda del contexto

Al proveer un mecanismo de traducción conveniente, se puede validar que las propiedades interesantes de B_A son satisfechas por una traducción de B_C.

Se puede decir que B_C es un refinamiento válido de B_A, dado el mecanismo de traducción T (con PI_A el conjunto de propiedades interesantes de A), si:

$$T(B_C) = < P_{TC}, VP_{TC}, BP_{TC} > \wedge (PI_A \subseteq BP_{TC} \subseteq VP_{TC})$$

Nótese que no es necesario que la traducción $T(B_C)$ se encuentre asociada a una descripción arquitectónica. Esto se debe a que una traducción es una consideración sobre la descripción arquitectónica como un todo y no sobre cada uno de los elementos en forma individual.

El problema es cómo definir el mecanismo de traducción T de forma tal que $T(B_C)$ sea lo "suficientemente parecida" a B_C como para creer que puede reemplazarlo al momento de verificar las propiedades. Si no se establece algún tipo de restricción, dados dos comportamientos cualquieras $B_A = < P_A, VP_A, BP_A >$ y $B_C = < P_C, \mathrm{VP_C}, BP_C >$ se podría definir el mecanismo $T(B_C) = B_A$ que determina un refinamiento válido para todo par de comportamientos. El desafío es encontrar un conjunto de restricciones que sean lo suficientemente débiles como para no impactar negativamente en la aplicabilidad del concepto y lo suficientemente fuerte para evitar "abusos" en el uso de las traducciones. En la siguiente sección se delinea una solución a este problema basada en el uso de estilos arquitectónicos.

3.4 Metodología de refinamiento con estilos

El problema de definir un mecanismo de traducción adecuado, puede resolverse a partir de la consideración de los estilos arquitectónicos que implementan cada una de las descripciones arquitectónicas que serán parte de la relación de refinamiento. El uso de estilos arquitectónicos es el método por excelencia de reutilización de conocimiento en el contexto de las descripciones arquitectónicas. La adopción de un estilo determinado, dota implícitamente a la descripción de una arquitectura de una serie de propiedades que caracterizan al estilo. Por ejemplo, un sistema que exhibe el estilo *Client-Server* tendrá por propiedad que la comunicación entre un *Cliente* dado y el *Servidor* sólo será iniciada por dicho *Cliente*.

Las descripciones utilizan diferentes estilos para lograr sus objetivos (exhibir ciertas propiedades), pero estos objetivos no pueden ser alcanzados únicamente mediante la adopción de uno o más estilos. El problema es que un estilo no define instancias de componentes y conectores [HNS99] en una descripción, sino qué predica sobre los roles y relaciones permitidos en la misma [CBB+02]; un estilo por sí mismo no define un comportamiento completo. Por ejemplo, el estilo *Client-Server* no dictamina que acciones desarrolla el *Server* ante cada *Llamada* o los motivos por los cuales el *Cliente* realiza dichas *Llamadas*.

La forma en que las descripciones instancian (utilizan) los estilos determina el comportamiento exhibido por las mismas. Así, los estilos permiten proveer el contexto necesario para el desarrollo de los conceptos propios del sistema, ya sea proveyéndolos explícitamente a través de las propiedades impuestas (En el ejemplo de la introducción, se usa el estilo *Announcer-Listener*

porque agencia de noticias, noticias y redacciones se mapean naturalmente a *Announcer*, *Eventos* y *Listeners*) o ayudando a que se manifiesten más fácilmente, al poder considerar las propiedades aportados como precondiciones válidas del sistema. En cualquiera de los casos se da el fenómeno de economía cognitiva [Liu00].

El uso de estilos permite particionar el conjunto de propiedades exhibidas por una descripción en dos subconjuntos disjuntos:

- **Propiedades de estilo:** Las propiedades que las descripciones exhiben implícitamente al utilizar los estilos

- **Propiedades de sistema:** Las propiedades "propias del sistema", quedan determinadas por la forma en que se instancia cada uno de los estilos utilizados y como éstos se combinan

En realidad, la partición presentada está precedida por una partición equivalente del lenguaje de la descripción, que resulta de considerar a los estilos como gramáticas:

- **Lenguaje de estilo:** El lenguaje aportado por los estilos a partir de los conceptos definidos por cada uno de ellos. Por ejemplo, el estilo *Client-Server* aporta los términos *Cliente*, *Servidor*, *Llamada* y *Respuesta*.

- **Lenguaje de sistema:** El lenguaje propio del sistema, permite definir la forma que se instancian y combinan los estilos. Por ejemplo, el término que describe la acción que el *Servidor* toma ante cada *Llamada* recibida.

Se puede considerar a los estilos como herramientas que son utilizadas para exhibir propiedades de un núcleo de conceptos propios del sistema. Este núcleo define la forma en que los estilos se instancian e interactúan entre ellos en aras de exhibir las propiedades de interés. Volviendo al ejemplo del estilo *Client-Server*, el núcleo conceptual del sistema define cuándo se realiza una *Llamada* y como se determina la *Respuesta* para cada una de ellas. De aquí que se pueda representar una descripción arquitectónica cómo en la siguiente figura:

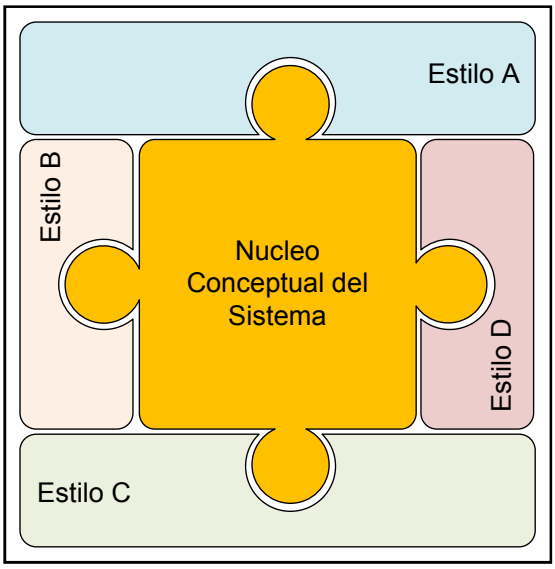

Figura 3-3: Uso de estilos arquitectónicos en las descripciones

Los estilos junto al núcleo conceptual, definen el sistema. Los estilos aportan conceptos y propiedades que el núcleo necesita para desarrollar y exhibir los conceptos propios del sistema.

Dadas dos descripciones de un mismo sistema, ambas compartirán ciertos conceptos de sus núcleos conceptuales porque deberían exhibir un mismo conjunto de "propiedades interesantes". Sin embargo, diferirán en los aspectos que atañen al nivel de abstracción como resultado del uso individual o en conjunto de diferentes estilos: La diferencia entre las descripciones serán las herramientas utilizadas por sendos núcleos para lograr sus cometidos. La tarea de refinamiento conlleva cambiar herramientas que exhiben cierto nivel de abstracción, por otras que exhiben un nivel de abstracción más adecuado. Si los estilos son las herramientas utilizadas por las descripciones, entonces se puede considerar que **refinar es equivalente a cambiar los estilos utilizados en una descripción**. El mecanismo de traducción deberá servir para determinar si las nuevas herramientas pueden suplantar a las anteriores.

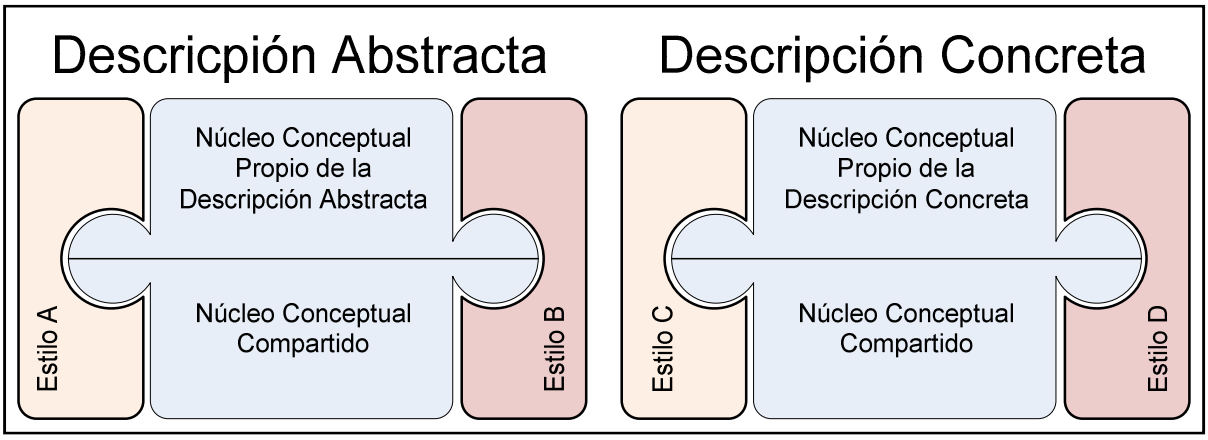

Figura 3-4: Relación entre descripciones de un mismo sistema

La propuesta es que el mecanismo de traducción sea una función de abstracción que abstraiga los estilos utilizados en la descripción más concreta junto a su núcleo conceptual propio hacia los estilos utilizados en la descripción más abstracta. Lo que se intenta es representar los estilos (i.e. las herramientas utilizadas) de la descripción más abstracta en función de los estilos y el núcleo conceptual propio de la descripción concreta. El núcleo propio de la descripción concreta se incluye ya que, tal vez, el estilo A de la descripción abstracta se puede representar solamente mediante la combinación de los estilos C y D de la descripción concreta; la combinación de dos estilos es parte del núcleo conceptual de una descripción

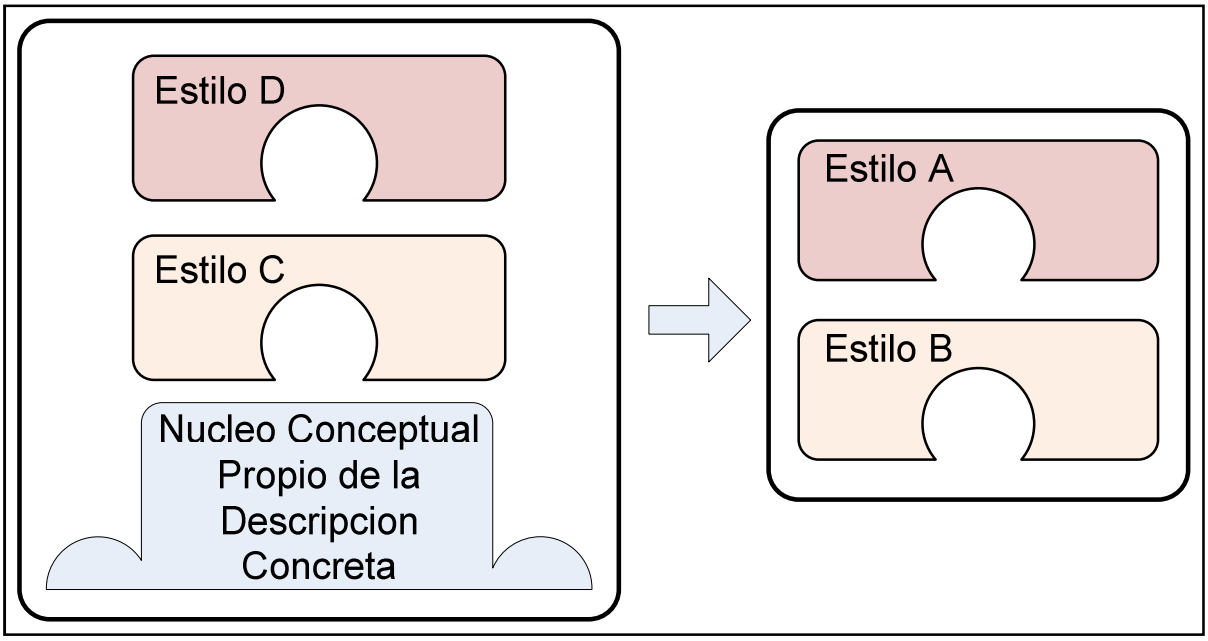

Figura 3-5: Mecanismo de traducción en base a estilos arquitectónicos

Esta función de abstracción se podría caracterizar a partir de la partición de los lenguajes de las descripciones determinado por el uso de estilos. Se puede reformular el mecanismo de traducción como una traducción del lenguaje $\alpha(Descripcion\ Concreta) - \alpha(Descripcion\ Abstracta)$ al lenguaje $\alpha(Estilos\ utilizados\ en\ la\ descripcion\ abstracta)$ donde α es la función "lenguaje de".

La presentación de la metodología incurre en ciertas liviandades para evitar definiciones que impliquen restricciones sobre las herramientas que se pueden utilizar. La idea es que lo aquí presentado, sirva como base para definir metodologías de refinamiento con estilos utilizando diferentes herramientas. A partir de éstas, se terminarán de definir los conceptos y el mecanismo de traducción adecuadamente. A estas metodologías se las llamará instanciaciones de la metodología de refinamiento con estilos.

Resumen de los conceptos utilizados por la metodología

De acuerdo a lo presentado hasta aquí, existen cuatro conceptos en el marco de estudio que sustenta la metodología:

- Proceso: De acuerdo a la interpretación de la vista conceptual adoptada, cada componente y conector de una descripción arquitectónica es un proceso independiente. Se debe definir la forma en que un proceso se describe, su semántica y como interactua con otros procesos

- Propiedad: Se debe definir la forma en que las propiedades se expresan (a partir de la definición de lenguaje propuesta) y como se determina la validez de las mismas para una descripción arquitectónica dada

- Mecanismo de Traducción: Se debe definir el mecanismo de traducción, respetando la definición de lenguaje propuesta y las restricciones impuestas por la metodología

- Lenguaje: Se debe definir una noción de lenguaje que permita expresar propiedades y proveer mecanismo de traducción apropiados

Una instanciación de la metodología deberá definir cada uno de los conceptos enumerados previamente mediante herramientas adecuadas para tal fin. En el siguiente capítulo se instancia la metodología propuesta mediante el uso del model checker LTSA y se presentan varios ejemplos de su aplicación práctica. Lo que resta de éste capítulo es una discusión acerca de los diferentes aspectos teóricos detrás de la propuesta presentada, mediante la cual se intenta justificar lo presentado en esta sección. Su lectura no es necesaria para la comprensión del resto del trabajo, pero ciertamente aporta a la construcción del concepto.

3.5 Discusión y justificación de la metodología propuesta

Para el desarrollo de la metodología, se toma como punto de partida tres conceptos fundamentales propuestos en los trabajos presentados en la sección “Nociones anteriores de refinamiento”:

1. Una definición de refinamiento que es relativa al sistema que se refina y a los motivos por los cuales se lleva a cabo la tarea de refinamiento;

2. La utilización de estilos arquitectónicos como medio de reutilización de conocimiento y experiencia comunitaria;

3. El uso de un mecanismo de traducción que permita formalizar como debe interpretarse una descripción a partir de otra.

Una definición relativa al sistema refinado

En los trabajos citados y en secciones anteriores de esta tesis, se hace referencia (en diferentes términos y por diferentes causas) a la “razón de ser” de un modelo arquitectónico. Moriconi y Qian lo llaman *raison d’etre*, Garlan “Propiedades de interés” y en el presente se lo denomina “espíritu”. En los tres casos se reconoce que detrás de toda descripción arquitectónica que se

realice, hay una serie de propiedades que se desea exhibir para comunicar, analizar, especificar y/o simular: Las descripciones existen para exhibir propiedades. El desacierto de Moriconi es NO reconocer que esta serie de propiedades no es necesariamente igual al conjunto de TODAS las propiedades que exhibe una descripción arquitectónica. Por su parte, el acierto de Garlan es definir la validez de un refinamiento en función de estas "Propiedades de interés" al reconocer que Moriconi se equivoca al enunciar su presupuesto de completitud. La necesidad de contar con un medio para reconocer si una descripción arquitectónica satisface o no una propiedad es central en todos los casos. Es por esto que la primera tarea que debe llevarse a cabo para obtener el resultado deseado en este trabajo, es dotar a las descripciones arquitectónicas de un significado o semántica que permita a un actor secundario determinar si ésta satisface o no una propiedad dada.

La semántica que se adopta para la metodología se basa en la propuesta de [HNS99] para la vista conceptual: El comportamiento de cada componente y conector se encuentra determinado por un proceso asociado que es independiente de los procesos de los demás elementos. El comportamiento de una descripción arquitectónica es el resultado de la interacción de todos estos procesos. A partir del comportamiento exhibido es que se determina que propiedades son satisfechas por la descripción.

Una vez definida la semántica adoptada, se propone analizar y clasificar las diferentes propiedades que pueden ser exhibidas por una descripción arquitectónica. De acuerdo a los principios planteados en la sección "¿Por qué es necesario refinar?" se considera que una arquitectura (y en particular cualquier descripción arquitectónica de la misma, que es el medio por el cual ésta se materializa) es un modelo parcial de un sistema, una representación que exhibe un nivel de abstracción determinado. Éste modelo satisface un número (posiblemente infinito) de propiedades, entre las cuales se encuentra un subconjunto (posiblemente acotado) de propiedades que justifican la existencia del modelo. Reformulando, la arquitectura existe para poder presentar ciertas propiedades que la implementación final del sistema debe poseer y que es de interés comunicar, analizar, especificar y/o simular. Reconociendo que no es posible exhibir todas las propiedades de interés de un sistema en forma simultánea (Porque los sistemas son necesariamente complejos [Brooks87], porque las propiedades pueden ser de naturaleza ortogonal y deben expresarse de diferentes maneras o por innumerables otras razones) es que se elige un nivel de abstracción y una herramienta de representación acorde, en aras de lograr el mayor detalle y la menor ambigüedad posible para la exhibición clara de ciertas propiedades. Como resultado existen ocasiones en las que del modelo final se pueden inferir, también, ciertas propiedades que no se intentaban comunicar en primer lugar y que se manifiestan como "efectos secundarios" de la elección anteriormente citada. A partir de esta "falencia fundamental" de la actividad de modelado, es que cobra importancia el contexto en el que el modelo es creado (incluyendo las razones por las cuales es creado). Toda descripción arquitectónica exhibirá entonces, dos conjuntos disjuntos de propiedades: las "propiedades interesantes" y las "propiedades accidentales", el contexto determinará la pertenencia de cada una de las propiedades a sendos grupos.

La definición de refinamiento presentada al intentar justificar la existencia del concepto en la primera sección de este capítulo, se basa en la idea de conservar el "espíritu" de una primera descripción en una segunda. Para la metodología presentada el "espíritu" de una descripción arquitectónica se corresponde con el conjunto de las "propiedades interesantes", y por lo tanto, también es relativo al contexto de ésta. El concepto de refinamiento presentado puede redefinirse entonces como:

La obtención de una nueva descripción arquitectónica como resultado de la reconsideración del nivel de abstracción de una descripción original, en aras de proveer un mayor grado de detalle para ciertos aspectos de interés. La nueva descripción debe satisfacer las "propiedades interesantes" de la primera

Utilización de estilos arquitectónicos

La actividad de modelado de una arquitectura se puede beneficiar (y generalmente lo hace) del conocimiento y la experiencia de la comunidad que la practica. En el marco de trabajo arquitectónico, la reutilización de conocimiento se manifiesta a partir de la adopción de uno o más estilos arquitectónicos. Mediante el uso de estilos, se dota implícitamente a las descripciones arquitectónicas de una serie de propiedades que hacen al *rationale* del estilo (Por ejemplo, el uso del estilo *Announcer-Listener* implica un patrón de comunicación entre componentes por el cual los componentes que cumplen el rol de *Listeners* son de naturaleza pasiva y el componente que cumple el rol de *Announcer* es de naturaleza activa). Al identificar los conceptos del sistema con los conceptos de los estilos utilizados, es que se aprovecha la existencia de las propiedades que estos proveen. Para el sistema presentado en la introducción de este informe, se pueden identificar a las redacciones periodísticas de los diarios como *Listeners*, a la agencia de noticias como un *Announcer* y a las noticias a publicar como los *Eventos* generados por éste último. Al utilizar el estilo *Announcer-Listener* para modelar el sistema (y mediante una interpretación acorde) se puede garantizar que todas las noticias emitidas por la central de noticias serán recibidas por las redacciones periodísticas suscriptas.

Un estilo no define instancias de componentes y conectores [HNS99] en una arquitectura, sino que predica sobre sus roles y relaciones [CBB+02]. Esto quiere decir que junto a las propiedades de una descripción arquitectónica resultantes de la aplicación de uno o más estilos, se encontrarán propiedades propias (valga la redundancia) de los componentes y los conectores definidos en la arquitectura, que determinan la forma en que los estilos aplicados fueron instanciados. En el sistema de noticias de la introducción, el hecho de que la comunicación entre el *Announcer* y los *Listeners* es iniciada por el primero, es una propiedad surgida de la utilización del estilo *Announcer-Listener*; el hecho de que cada *Listener* (es decir, cada redacción) genere un aviso en pantalla en respuesta a cada *Evento* (noticia) recibido es una propiedad propia del componente que cumple el rol de *Listener*. Se puede realizar, entonces, una segunda clasificación del conjunto de propiedades exhibidas por una descripción arquitectónica que resulta en dos particiones: el subconjunto de "propiedades de los estilos utilizados" y el subconjunto de "propiedades propias del sistema". Esta clasificación es ortogonal a la anteriormente presentada (Los nombres de los

subconjuntos se abreviarán como “Propiedades de estilos” y “Propiedades de sistema” para comodidad del lector).

Existe una diferencia fundamental entre las clases presentadas anteriormente y es que las “propiedades de sistema” se deben enunciar explícitamente, ya que definen la forma en que los estilos utilizados fueron instanciados. En contraposición, no es necesario enunciar las “propiedades de estilos” ya que estas forman parte del conocimiento público y son las mismas para todos los sistemas que hagan uso de dichos estilos. Es necesario aclarar que en numerosas oportunidades, además de elegir un estilo, se elige un sub-estilo del mismo (Dado que un sub-estilo es una especialización de un estilo dado, puede aportar un número mayor de propiedades y conceptos a la descripción que el estilo especializado). La elección de un sub-estilo no implica que todas sus propiedades vayan a ser, necesariamente, miembros del conjunto de “propiedades interesantes” del modelo, pero sí debería implicar que una gran parte de ellas lo sean. Mientras mayor sea el número, más efectiva habrá sido la elección de dicho sub-estilo al maximizar la reutilización del conocimiento comunal que éste representa (y por ende minimizar la necesidad de generar nuevo conocimiento, dando lugar al fenómeno de economía cognitiva [Liu00]). En el ejemplo presentado la economía cognitiva se manifiesta al lograr garantizar el envío y recepción de noticias en forma “gratuita”, al utilizar un estilo adecuado e interpretar sus conceptos convenientemente.

El uso de estilos arquitectónicos y la consecuente partición de propiedades presentada supone dos fenómenos de interés que permiten caracterizar a las propiedades de una descripción arquitectónica: El primero es que las “propiedades de sistema” se enuncian más fácilmente ya que se pueden formular a partir del hecho de que las “propiedades de estilo” se suponen ciertas (es decir, se pueden considerar como pre-condiciones válidas). En el caso de las noticias, se puede enunciar como propiedad “Cada vez que se recibe una noticia se muestra una alerta en pantalla” que es mucho más simple que “Cada vez que la agencia de noticias emite una noticia, en cada diario que haya enviado previamente una petición de suscripción que todavía no fue cancelada se muestra una alerta en pantalla”. El segundo es que los estilos definen un lenguaje a partir del cual se pueden expresar las propiedades exhibidas por una descripción. Utilizando nuevamente como ejemplo el estilo *Announcer-Listener*, se pueden expresar las propiedades de una descripción que lo utilice, en términos de *Listeners*, *Announcers*, *Eventos* y *Suscripciones*. De ésta forma, se puede definir el lenguaje de una descripción como la unión de todos los lenguajes generados por los estilos aplicados, junto al lenguaje propio del modelo. El lenguaje de la descripción del sistema de noticias, incluiría los términos del estilo *Announcer-Listener* junto a los términos propios del sistema (Por ejemplo, *MostrarAlertaNoticia* como representación de la acción de mostrar la alerta de una noticia recibida en pantalla). La propiedad de sistema presentada anteriormente se puede reformular como: “Por cada *Evento* recibido el *Listener* procede a *MostrarAlertaNoticia”;* que en cristiano significa: Por cada noticia recibida la redacción periodística del diario emite una alerta en la pantalla (Que gracias al estilo utilizado implica que por cada noticia emitida por la central de noticias las redacciones periodísticas emiten una alerta en sus pantallas: Economía cognitiva en acción).

De los dos fenómenos citados, se puede establecer que la utilización de uno o más estilos en una descripción arquitectónica, tiene una relación directa con el nivel de abstracción que ésta exhibe. Al definir lenguajes y propiedades se afecta (positiva o negativamente) la capacidad de un modelo para exhibir las "propiedades de interés" a un nivel de detalle determinado. Llevando el concepto un paso más lejos, se puede argumentar que la tarea de refinar (que conlleva reconsiderar el nivel de abstracción en el que se trabaja) es equivalente a cambiar los estilos utilizados en una descripción, por otros que aporten un grado de detalle mayor.

Resumiendo lo expuesto anteriormente, los estilos se pueden considerar como una gramática: la definición de un lenguaje junto a las reglas para la utilización del mismo. Es así, que los estilos juegan tres roles fundamentales en el marco de trabajo arquitectónico en general y en esta metodología en particular:

1. Dotar, implícitamente, a las descripciones arquitectónicas de una serie de propiedades;

2. Presentar un lenguaje a partir del cual se pueden expresar las propiedades exhibidas;

3. Simplificar la escritura de las propiedades, ya que éstas se pueden formular a partir de las premisas (ciertas) resultantes de las propiedades impuestas por el estilo.

Las descripciones arquitectónicas exhiben, mediante una semántica definida, un número (tal vez infinito) de propiedades. Un subconjunto de éstas, conforman el "espíritu" de la descripción. El "espíritu" es el conjunto de "propiedades interesantes" que el modelo intenta exhibir claramente y que justifican la creación del mismo. La claridad con la que se logre exhibir estas propiedades (y la forma en que son formuladas), queda determinada por la utilización de uno o más estilos arquitectónicos en la descripción. Las mejores descripciones, serán aquellas que logren destacar su "espíritu" por sobre las "propiedades accidentales".

Mecanismo de traducción como formalización de una interpretación

A primera vista, reconocer si un refinamiento conserva el "espíritu" de una descripción parece fácil: se debe probar si el refinamiento satisface las "propiedades interesantes" que conforman el "espíritu" original. Sin embargo, existe un hecho ineludible que atenta contra este planteo: El "espíritu" de la descripción original es exhibido en términos del lenguaje de la misma. El refinamiento, al ser una reconsideración del nivel de abstracción, posiblemente haga uso de diferentes estilos y, por lo tanto, su lenguaje sea diferente al de la descripción original. ¿Cómo se puede asegurar que el "espíritu" original es exhibido por el refinamiento si las propiedades que este exhibe se describen en un lenguaje diferente?

La respuesta obvia, es tratar de obtener un lenguaje común a la descripción concreta y a las "propiedades interesantes". Reformular ambos conceptos en términos de un lenguaje único no es una solución plausible ya que los lenguajes de sendos conceptos son el resultado de haber considerado un nivel de abstracción adecuado para ellos y, por lo tanto, son esenciales a los mismos. La solución propuesta es, entonces, proveer un mecanismo de traducción entre los

lenguajes; este formalizaría como se combinan y utilizan los términos de la descripción más concreta (es decir, como se deben interpretar) para obtener los términos de la descripción más abstracta. Retomando el ejemplo planteado en la introducción de este trabajo, la descripción más concreta hace uso del estilo *Cliente-Servidor* para proveer una funcionalidad similar a la propuesta por el estilo *Announcer-Listener* (y así mantener el "espíritu original", ya que en el contexto planteado no se cuentan con las herramientas adecuadas para implementar la solución a partir de la descripción original). Se propone interpretar la descripción más concreta de la siguiente manera:

1. El componente *Agencia de Noticas,* que en la descripción más concreta hace las veces de *Server,* representa al *Announcer* de la descripción más abstracta. Este mantiene una lista de *Eventos* pendientes para cada *Diario* que se encuentre suscripto. Anunciar un nuevo *Evento* es equivalente a agregar el mismo a las listas de pendientes.

2. Los componentes *Diario 1, 2 y 3*, que cumplen el rol de *Clientes* en la descripción más concreta, representan a los *Listeners* de la descripción más abstracta. Estos realizan llamados al *Server* para simular la suscripción y desuscripción a los *Eventos.* Si se encuentran "suscriptos", interrogan periódicamente al *Server* sobre la existencia de nuevos eventos. Si la lista de pendientes para dicho *Listener* no es vacía, el *Server* quitara el *Evento* que se encuentre al frente de la lista y éste será la respuesta. En caso contrario se responde informando que no existen eventos.

La interpretación propuesta es un mecanismo de traducción informal. Ésta hace las veces de una función de abstracción, por la cual se expresan los elementos de la descripción más abstracta a partir de los elementos de la descripción más concreta. Se puede pensar a la interpretación propuesta como un "filtro" o una "lupa", que aplicada a la descripción arquitectónica más concreta, permite visualizarla en términos del lenguaje más abstracto, eliminando los detalles "extras" aportados:

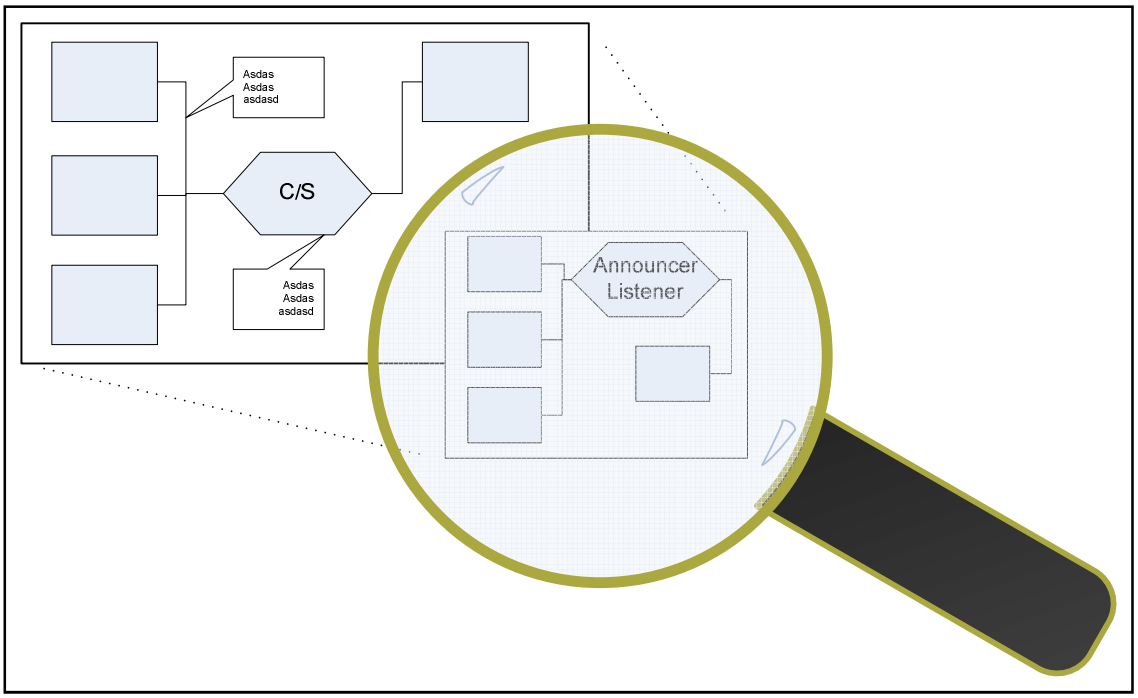

Figura 3-6: La interpretación elimina los detalles "extras" aportados por la descripción concreta

Si la descripción "filtrada" satisface las "propiedades de interés" (afirmación que ahora puede constatarse o rechazarse ya que la descripción "filtrada" se encuentra expresada en los mismos términos que éstas), entonces el "espíritu" de la descripción original se mantiene en la descripción más concreta de acuerdo al contexto dado.

Observando la interpretación propuesta, se ve que ésta se describe utilizando términos de los estilos de la descripción más concreta (*Cliente, Servidor, Llamadas*), términos "propios del sistema" de la descripción concreta (Listas de eventos pendientes) y términos de los estilos de la descripción más abstracta (*Eventos, Announcer, Listeners*). No se mencionan términos "propios del sistema" de la descripción abstracta, ni términos compartidos entre las dos descripciones (como podría ser *MostrarAlertaNoticia,* la representación de la acción de mostrar una alerta). Esto no es casual; es el resultado de una restricción que es necesaria imponer al mecanismo de traducción y que obedece a aspectos tanto prácticos como teóricos. En el aspecto teórico lo que se intenta demostrar es que el uso de cualquier estilo puede considerarse accidental y coyuntural: los conceptos desarrollados mediante dichos estilos son los que se busca preservar al refinar. Se considera que los términos del lenguaje "propio del sistema" de una descripción (y en particular de la descripción abstracta) se suponen ideales y solo pueden ser representados por sí mismos. Por su parte, los términos compartidos entre las dos descripciones no necesitan ser traducidos. Tampoco éstos pueden utilizarse para representar otros conceptos de la descripción abstracta, porque de esta forma estarían desempeñando un rol no exhibido en la descripción original (describir un segundo concepto de la misma).

En el aspecto práctico, la restricción es necesaria para evitar potenciales "abusos" en el uso del mecanismo: Los términos del lenguaje "propio del sistema" hacen las veces de invariante, que permite determinar la idoneidad de la interpretación al no ser afectados por la misma.

La propuesta es, entonces, proveer una traducción del lenguaje propio (no compartido) de la descripción más concreta al lenguaje de los estilos de la descripción más abstracta. Se adopta esta dirección en la traducción (y no la opuesta: traducir del lenguaje de la descripción más abstracta al lenguaje de la descripción más concreta) para evitar reformular las "propiedades de interés" en un nuevo lenguaje. Identificar cuáles son las características que se desean en un sistema, comprenderlas y expresarlas convenientemente es una tarea lo suficientemente compleja como para evitar realizarla repetidas veces; si las propiedades fueron expresadas en un lenguaje dado, es porque éste es el más adecuado para tal fin. De esta forma se obtiene una nueva definición del concepto de refinamiento:

Dada una descripción arquitectónica AD, refinar es el proceso de obtener una nueva descripción arquitectónica AD' como resultado de la reconsideración de los estilos utilizados en la descripción original y en aras de proveer un mayor grado de detalle para ciertos aspectos de interés. Junto a la nueva descripción se debe incluir una función de traducción T que traduzca desde el lenguaje propio de AD' (no compartido con AD) hacia el lenguaje de los estilos utilizados en AD. La traducción propuesta debe cumplir con las siguientes propiedades:

1. $T(AD') \vdash (Propiedades\ Estilos\ AD \cap Propiedades\ Interesantes\ AD)$
2. $T(AD') \vdash (Propiedades\ Sistema\ AD \cap Propiedades\ Interesantes\ AD)$

Se dice entonces que AD' es un refinamiento válido de AD

De acuerdo a lo expuesto en el párrafo anterior, un par de descripciones arquitectónicas por si solas no se pueden considerar un refinamiento (ni correcto, ni erróneo) ya que falta establecer un contexto que termine de definir la relación entre ambas. Mediante el cumplimiento de las propiedades enumeradas anteriormente (que son relativas al contexto, ya que involucra la forma en la que las propiedades quedan particionadas y al mecanismo de traducción propuesto) es que se puede afirmar que el refinamiento conserva el "espíritu" de la descripción original. La satisfacción de la primera propiedad es la que permite validar la interpretación de los estilos de la descripción original en términos de los estilos de la descripción más concreta. La satisfacción de la segunda propiedad permite validar si la instanciación de los estilos interpretados en la descripción más concreta es similar a la instanciación de los mismos en la descripción original (en cuanto a su utilización para la satisfacción de las "propiedades interesantes"). Un refinamiento es incorrecto si la traducción propuesta no resulta en el cumplimiento de alguna de las dos propiedades presentadas en la definición (y por lo tanto, en la no satisfacción de alguna de las "propiedades interesantes").

En el siguiente capítulo se instancia la metodología propuesta mediante el uso del model checker LTSA y se presentan varios ejemplos de su aplicación práctica, por los cuales se demuestran los alcances de la metodología y los diferentes aspectos que deben ser considerados para su uso.

4. Metodología de refinamiento con estilos utilizando LTSA

En este capítulo se presenta una instanciación completa de la metodología propuesta, utilizando el model checker LTSA. Se definen los conceptos enumerados en el capítulo anterior mediante ésta herramienta y se muestran varios ejemplos de aplicación de esta instanciación de la metodología (incluyendo la validación del ejemplo presentado en la introducción de este trabajo). En el desarrollo de este capítulo se intentará aportar claridad sobre los conceptos y principios teóricos de la metodología propuesta al exhibir ventajas, desventajas y otros aspectos prácticos asociadas a la aplicación de la misma.

LTSA [MK99] es una herramienta de verificación para sistemas concurrentes cuya funcionalidad consiste en verificar mecánicamente si la especificación de un sistema satisface, o no, las propiedades requeridas para el mismo. Los sistemas y sus propiedades se modelan en LTSA como un conjunto de máquinas de estado finito que interactúan entre sí. LTSA compone estas máquinas y realiza análisis de alcanzabilidad para buscar en forma exhaustiva violaciones de las propiedades por parte de la especificación del sistema. Formalmente hablando, cada una de las máquinas es un sistema de transiciones etiquetado (LTS) que define por completo todos los estados alcanzables y sus transiciones. Para facilitar la escritura de las especificaciones y propiedades, se utiliza un álgebra de procesos llamada FSP que LTSA interpreta y traduce convenientemente. A continuación se definen los conceptos centrales de la metodología a partir de las construcciones propias de LTSA, los mismos serán aplicados en los ejemplos presentados en este capítulo. Se recomienda al lector realizar una primera lectura de las definiciones y luego volver sobre las mismas durante el desarrollo de los ejemplos. El código fuente completo de todos los ejemplos (junto a una breve explicación de su implementación) puede encontrarse en el Apéndice B de este trabajo.

Procesos

Cada **proceso** de la metodología se identifica mediante un único proceso (simple o compuesto) FSP. La comunicación entre procesos se realiza mediante acciones comunes a dos o más procesos. Sea $LTSAProcessesDescriptions$ el conjunto universal de procesos descriptos mediante FSP, entonces para todo proceso p vale:

$$p \in LTSAProcessesDescriptions \subseteq ProcessesDescriptions$$

Propiedades

En el contexto de la instanciación LTSA, cualquiera de las construcciones enumeradas a continuación se considera una **propiedad**:

- Una propiedad de liveness LTSA,
- Una propiedad de safety LTSA,
- Una aserción expresada mediante Fluent Linear Temporal Logic (FLTL) [MK99],
- Un proceso LTSA que cumple con la propiedad $CustomProperty$

con *CustomProperty* definida de la siguiente forma:

$$CustomProperty(proc) \equiv \left(\forall e: states(LTS(proc)) - \{\pi\}, \forall a: \alpha_{FSP}(proc), \exists\, e': states\left(LTS(proc)\right) \middle| e \xrightarrow{a} e'\right) \wedge (\pi \in states\left(LTS(proc)\right))$$

donde:

- $LTS(proc)$ es una función que retorna el sistema de transiciones etiquetado representado por el proceso $proc$
- $states(l)$ es el conjunto de estados del sistema etiquetado de transiciones l
- $\alpha_{FSP}(proc)$ es el alfabeto del proceso $proc$
- $\boldsymbol{\pi}$ es el estado "error" [MK99]

Dado p_1 tal que $CustomProperty(p_1)$ y p_2 tal que $\alpha_{FSP}(p_1) \subseteq \alpha_{FSP}(p_2)$, entonces toda traza del proceso $(p_1 \parallel p_2)$ será también una traza de p_2 o el prefijo de una traza de p_2 (en este último caso, la traza original viola la propiedad definida por p_1).

Se llama *LTSAProperties* al conjunto que incluye a todos los elementos definidos a partir de las construcciones enumeradas anteriormente.

Lenguajes

Se define la función α que dado un proceso $proc$ una propiedad $prop$ retorna el **lenguaje** asociado de la siguiente forma:

$$\alpha(proc) = \alpha_{FSP}(proc)$$

$$\alpha(prop) = \begin{cases} \alpha_{FLTL}(prop)\ si\ prop\ es\ una\ asercion\ FLTL \\ \alpha_{FSP}(PropertyAsFSP(prop))\ en\ caso\ contrario \end{cases}$$

con

- α_{FSP} la función que retorna el alfabeto de un proceso FSP
- α_{FLTL} la función que retorna el alfabeto de una aserción expresada mediante FLTL
- $PropertyAsFSP$ la función que retorna el proceso FSP al que se reduce la propiedad para ser compuesta y validada en LTSA

El lenguaje de un conjunto de procesos P se define como:

$$\alpha(P) = \bigcup_{p \in P} \alpha(p)$$

Comportamientos y Comportamientos asociados

En el capítulo anterior se presenta la definición de comportamiento como una tripla $B =< P, VP, BP >$ con P un conjunto de procesos, VP el conjunto de propiedades que son verificables

con este comportamiento y $BP \subseteq VP$ el conjunto de propiedades que son satisfechas. En el contexto de la instanciación propuesta, los **comportamientos** se especializarán con las siguientes restricciones:

1. $P \subseteq LTSAProcessesDescriptions$
2. $prop \in VP \Leftrightarrow \alpha(prop) \subseteq \alpha(P)$
3. $prop \in BP \Leftrightarrow BehaviorAsFSP(B) \vdash prop$

donde,

4. $BehaviorAsFSP(P) \equiv ||_{proc \in P}\ proc$
5. $BehaviorAsFSP(< P, VP, BP >) \equiv BehaviorAsFSP(P)$

Como resultado, cada uno de los comportamientos propuestos en el marco de ésta instanciación deben definir únicamente el conjunto P, ya que los conjuntos VP y BP quedan determinados a partir del primero.

Dado un modelo de comportamiento $B = < P_B, VP_B, BP_B >$, se define el lenguaje de B como:

$$\alpha(B) = \alpha(P_B)$$

Un **comportamiento asociado a una descripción arquitectónica** $AD =< CN, CX, P, R, \overset{\boldsymbol{R}}{\rightarrow}, \overset{\boldsymbol{P}}{\rightarrow}, \overset{\boldsymbol{AD}}{\leftrightarrow} >$ es un comportamiento $B_{AD} =< P_{AD}, VP_{AD}, BP_{AD} >$ tal que existe una función ψ que relaciona los procesos de B_{AD} con los componentes y conectores de AD. Para evitar interacciones prohibidas entre componentes y conectores de la arquitectura (a saber: entre dos componentes, entre dos conectores o entre un conector y un componente que no se encuentren asociados en la descripción) la función ψ debe cumplir con las siguiente propiedad:

$$CommunicationIsLegal(\psi) \equiv (\forall\, m1, m2 \in Dom(\psi) | m_1 \neq m_2 \Rightarrow CommunicationIsLegal(m1, m2))$$

con:

$$CommunicationIsLegal(m_1, m_2) \equiv \alpha(\psi(m_1)) \cap \alpha(\psi(m_2)) \neq \emptyset$$
$$\Rightarrow \left(\left(m_1 \overset{AD}{\leftrightarrow} m_2 \right) \right.$$
$$\left. \vee \left(\exists pt \in paths(m_1, m_2), \forall\, m_i \in pt \right) \middle| \alpha(\psi(m_1)) \cap \alpha(\psi(m_2)) \subseteq \alpha(\psi(m_i)) \right) \Big)$$

donde $paths(m_1, m_2)$ es la función que devuelve el conjunto de caminos por los cuales los elementos m_1 y m_2 se encuentran asociados. Un camino entre m_1y m_2 es una lista de componentes y conectores $[m_{i0}, m_{i1}, \dots, m_{in}]$ tales que

$$\left(\forall j : 0..n-1 \middle| m_{ij} \overset{AD}{\leftrightarrow} m_{ij+1} \right) \wedge (m_1 \overset{AD}{\leftrightarrow} m_{i0}) \wedge (m_{in} \overset{AD}{\leftrightarrow} m_2)$$

Estilos

Un **estilo** es, en el contexto de esta instanciación, una tupla de la forma

$$S = < Props_{S,}\ Configuration_S >$$

con:

1. $Props_S \subseteq LTSAProperties$
2. $Configuration_S = \{cf_1, \dots, cf_n\}$

donde

1. $cf_i \subseteq \alpha(Props_S)$

Un estilo consiste de un número determinado de propiedades, que definen implícitamente el lenguaje del estilo y de una partición de dicho lenguaje. Un comportamiento B_{AD}, asociado a una arquitectura $AD =< CN, CX, P, R, \overset{\boldsymbol{R}}{\rightarrow}, \overset{\boldsymbol{P}}{\rightarrow}, \overset{\boldsymbol{AD}}{\leftrightarrow} >$ mediante la función ψ, exhibe el estilo S utilizando la traducción canónica T_S si:

1. $BehaviorAsFSP_{T_S}(B_{AD}) \vdash\ Props_S$
2. $\forall cn \in CN, T_S(\psi(cn)) \cap cf_i \neq \emptyset \Rightarrow \left(\forall j: 1..n, j \neq i \Rightarrow T_S(\psi(cn)) \cap cf_j = \emptyset\right)$

La inclusión de las traducciones canónicas responde a un tecnicismo que será presentado y resuelto en el ejemplo presentado en la próxima sección. Por lo pronto, se puede considerar que toda traducción canónica es igual a la función identidad y que $BehaviorAsFSP_{T_S}(B_{AD}) = BehaviorAsFSP(B_{AD})$. La intención detrás del elemento $Configuration_S$ de la tupla es que cada componente utilice únicamente los términos del lenguaje del estilo correspondiente a su rol (Por ejemplo, que un *Listener* no emita un *Anuncio*).

Dado el estilo $S = < Props_{S,}\ Configuration_S >$, se define el lenguaje de S como:

$$\alpha(S) = \bigcup_{prop\ \in\ Props_S} \alpha(prop)$$

Mecanismos de traducción

Un **proceso de traducción** de un lenguaje origen α_A a un lenguaje destino α_B es un proceso FSP que cumple la siguiente propiedad:

$$TranslationProcess_{\alpha_A,\alpha_B}(proc)$$
$$\equiv\ (\forall e: states(LTS(proc)) \mid SimpleState(proc, e) \vee\ TranslationState(proc, e))$$
$$\wedge\ (\alpha(proc) =\ \alpha_A\ \cup \alpha_B)$$

con:

- $SimpleState(proc, e) \equiv$
  $\left(\forall a: \alpha_A, \exists\ e': states\left(LTS(proc)\right) \middle| e \xrightarrow{a} e'\right)\ \wedge \left(\nexists b: \alpha_B, \exists\ e': states\left(LTS(proc)\right) \middle| e \xrightarrow{b} e'\right)$

- $TranslationState(proc, e) \equiv$
  $\left(\exists\, b{:}\,\alpha_B, \exists\, e'{:}\,states(LTS(proc)) \middle| e \xrightarrow{b} e' \wedge \left(\forall a{:}\,\alpha(proc) - \{b\}, \nexists\, e''{:}\,states(LTS(proc)) \middle| e \xrightarrow{a} e''\right)\right)$

Un conjunto de procesos de traducción $\{tp1_{\alpha A,\alpha B1}, \ldots, tpn_{\alpha A,\alpha Bn}\}$, se puede considerar un **mecanismo de traducción** de un comportamiento B_{AD2} a un comportamiento B_{AD1} si se respetan las siguientes restricciones propuestas por la metodología:

1. $\alpha A \subseteq \alpha(B_{AD2}) - \alpha(B_{AD1})$
2. $\alpha B_i \subseteq \bigcup_{S_i \in Estilos\ utilizados\ en\ B_{AD1}} \alpha(S_i)$

En la práctica la restricción número dos se remplaza por:

2. $\alpha B_i \subseteq \bigcup_{T_i \in Traducciones\ canónicas\ para\ B_{AD1}} LangUsedToImplement_{T_i}(B_{AD1})$

Las definiciones necesarias para comprender este remplazo se postergan hasta la presentación del primer ejemplo de refinamiento con LTSA en la próxima sección, pero básicamente la nueva restricción indica que el lenguaje de destino de los procesos de traducción no debe ser el de los estilos utilizados en los comportamientos, sino que el lenguaje de destino debe ser el utilizado para *implementar* los estilos exhibidos en el comportamiento B_{AD1}.

El resultado de aplicar un mecanismo de traducción MT al comportamiento $B_{AD2} = < P_{AD2}, VP_{AD2}, BP_{AD2} >$ es la obtención de un nuevo comportamiento

$$MT(B_{AD2}) = < P_T, VP_T, BP_T >$$

con:

$$P_T = \{ \left(BehaviorAsFSP(B_{AD2}) \parallel tp1_{\alpha A,\alpha B1} \parallel \ldots \parallel tpn_{\alpha A,\alpha Bn}\right) \ll \bigcup_{i=1..n} \alpha Bi \}$$

y los conjuntos VP_{AD2} y BP_{AD2} quedan definidos a partir de éste, como fuera explicado anteriormente. Dado $MT(B_{AD}) = < \{p_t\}, VP_T, BP_T >$, se define

$$MTasFSP(B_{AD}) = p_t$$

Un comportamiento determina el conjunto de todas las trazas de acciones que puede producir la ejecución de un sistema. Dichas trazas estarán conformadas por las acciones incluidas en los alfabetos de los diferentes procesos FSP que son compuestos para simular la ejecución concurrente. Al aplicar un mecanismo de traducción tal como se definió anteriormente, se está "decorando" las trazas del sistema original con las acciones de los lenguajes destino de los procesos de traducción del mecanismo. Por "decorar" se entiende a la inserción de acciones en puntos específicos de las trazas, el orden entre las acciones originales de las trazas se mantiene inalterado. En el ejemplo que se encuentra a continuación, $Trace_1$ es una traza de un proceso

cuyo alfabeto contiene a las acciones $\{a, b, c\}$ y $Trace_2$ es una "decoración" de $Trace_1$ con las nuevas acciones $\{b, e\}$

$$Trace_1 = \{a, b, c, a, b, c, b, c, a, b, c, a, \ldots\}$$

$$Trace_2 = \{a, b, c, a, b, \boldsymbol{d}, c, b, c, \boldsymbol{e}, a, b, c, \boldsymbol{e}, a, \ldots\}$$

De esta forma se obtiene un nuevo comportamiento "muy parecido al original", que amplía el lenguaje del mismo al incluir acciones de un segundo comportamiento. Se dice que es "muy parecido al original" gracias al cumplimiento de la siguiente propiedad:

Sea $B_{AD} = < P_{AD}, VP_{AD}, BP_{AD} >$ un comportamiento y $TB_{AD} = < P_T, VP_T, BP_T >$ una traducción obtenida mediante un mecanismo de traducción entonces, para toda propiedad $p \in VP_{AD}$

1. $p \in VP_{AD} \Rightarrow p \in VP_T$
2. $(p \in BP_{AD} \nLeftrightarrow p \in BP_T) \Rightarrow$
   $\left(\exists p', \forall proc \middle| \left(\alpha(proc) = \alpha(B_{AD})\right) \Rightarrow (proc \vdash p' \Leftrightarrow proc \vdash p) \wedge (p' \in BP_{AD} \Leftrightarrow p' \in BP_T)\right)$

Es decir, toda propiedad verificable en B_{AD} es verificable en TB_{AD} y si ocurriera que una propiedad verificable en B_{AD} es válida para B_{AD} pero no para TB_{AD}, ésta puede reemplazarse por una propiedad equivalente que sea válida o inválida para ambos comportamientos. Esto es posible ya que la invalidez de la propiedad en TB_{AD}, es el resultado de no considerar el lenguaje ampliado.

Refinamiento válido

Dada una descripción arquitectónica AD_1 con un comportamiento asociado B_1 tales que:

1. AD_1 exhibe los estilos $S_1 \ldots S_n$
2. P_{Si} es el conjunto de propiedades del estilo S_i que son satisfechas por B_1
3. T_{Si} es la traducción canónica tal que $BehaviorAsFSP_{T_{Si}}(B_1) \vdash P_{Si}$
4. PI_{AD1}es el conjunto de "propiedades interesantes" de sistema
5. $BehaviorAsFSP(B_1) \vdash PI_{AD1}$

Entonces la descripción arquitectónica AD_2 con un comportamiento asociado B_2 es un refinamiento válido para el par $AD_1; B_1$ de acuerdo al mecanismo de traducción MT si:

1. $MTasFSP(B_2) \vdash PI_{AD1}$
2. $T_{Si}\left(MTasFSP(B_2)\right) \vdash P_{Si}$

La primera propiedad implica que todas las propiedades interesantes de sistema de B_1 deben ser satisfechas (mecanismo de traducción mediante) por B_2. La segunda propiedad implica que todas las propiedades interesantes de estilo de B_1 deben ser satisfechas (mecanismo de traducción y traducciones canónicas mediante) por B_2.

Como se aclara en el punto anterior, La inclusión de las traducciones canónicas responde a un tecnicismo que será presentado y resuelto en el ejemplo presentado en la próxima sección. Por lo

pronto, se puede considerar que toda traducción canónica es igual a la función identidad y que $BehaviorAsFSP_{T_{Si}}(B_1) = BehaviorAsFSP(B_1)$

4.1 Validación de refinamiento: Agencia de noticias

En esta sección se utilizan los conceptos presentados anteriormente para validar el ejemplo de la agencia de noticias del primer capítulo de este trabajo. Como primer paso se presenta una descripción arquitectónica formal basada en la del ejemplo y un comportamiento asociado para la misma. A partir de aquí, las descripciones arquitectónicas se presentarán mediante un diagrama que debe interpretarse de acuerdo a lo expuesto en la sección "Marco teórico-formal para el concepto de refinamiento" del capítulo anterior. Los comportamientos asociados a las descripciones se presentarán mediante la combinación de código FSP explícito y la definición por extensión de la función ψ que relaciona elementos arquitectónicos con sus procesos (ya que estos dos elementos son suficientes para determinar el comportamiento asociado a la descripción). El código fuente completo para este ejemplo se puede encontrar en los listados `NewsService_AnnouncerListener.lts` y `NewsService_ClientServer.lts` del Apéndice B.

Descripción y comportamiento del sistema

Para comenzar con el ejemplo, se presenta la descripción arquitectónica del sistema propuesto (a la que se llamará $AN1$ o "descripción abstracta") mediante el diagrama que aparece a continuación:

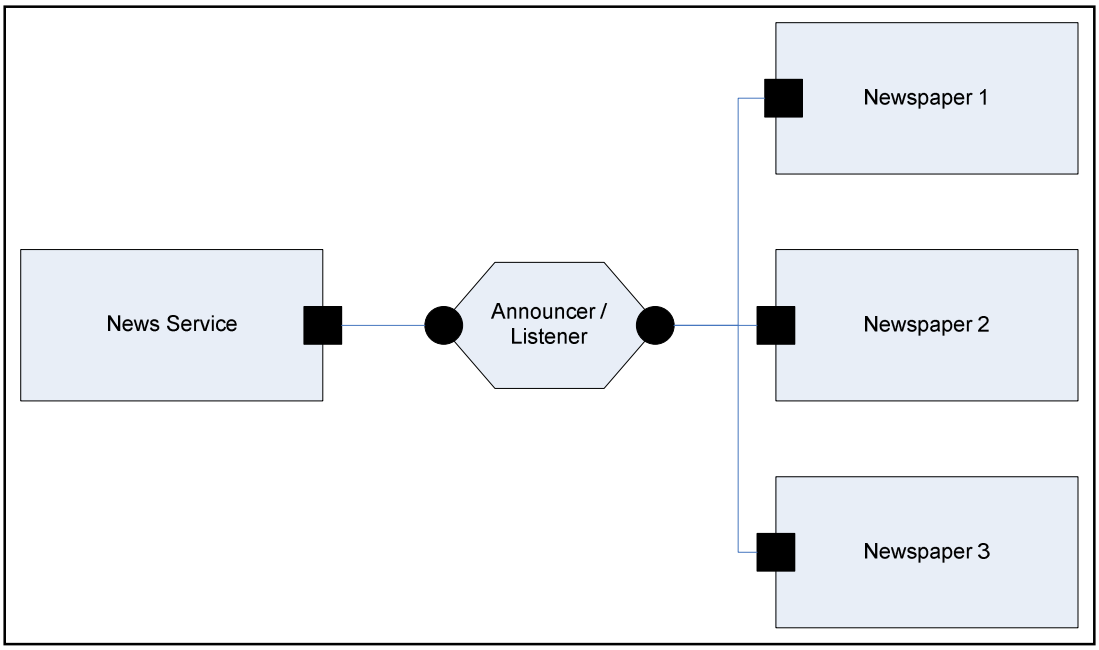

Figura 4-1: Descripción del sistema de agencia de noticias utilizando el estilo Announcer-Listener

Se provee también el siguiente código FSP, que se utilizará para definirá la función ψ_{AN1} a partir de la cual se determina el comportamiento asociado a la descripción $AN1$

```
const NEWSPAPER_MIN = 1
const NEWSPAPER_MAX = 3
range NEWSPAPERS = NEWSPAPER_MIN..NEWSPAPER_MAX

const NEWS_MIN = 1
const NEWS_MAX = 2
const NEWS_A = NEWS_MIN
const NEWS_B = NEWS_MAX
range NEWS = NEWS_A..NEWS_B

// NEWS_SERVCE
// -----------
NEWS_SERVICE = (newsRedacted[n:NEWS] -> sendNewsAlert[n] -> NEWS_SERVICE).

// NEWSPAPER
// ---------
NEWSPAPER =
(
       subscribe -> NEWSPAPER_SUBSCRIBED
),
NEWSPAPER_SUBSCRIBED =
(
       newsAlertReceived[n:NEWS] -> newsAlertOnScreen[n] -> NEWSPAPER_SUBSCRIBED
       | unsubscribe -> NEWSPAPER
).

|| NEWSPAPER_1 = (newspaper[1]:NEWSPAPER).

|| NEWSPAPER_2 = (newspaper[2]:NEWSPAPER).

|| NEWSPAPER_3 = (newspaper[3]:NEWSPAPER).

// CONNECTOR
// ---------
CONX_NEWSPAPER_QUEUE(N=NEWSPAPER_MIN) =
(
       sendNewsAlert[NEWS] -> CONX_NEWSPAPER_QUEUE
       | newspaper[N].subscribe -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED
),
CONX_NEWSPAPER_QUEUE_SUBSCRIBED =
(
       sendNewsAlert[d:NEWS] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[d]
       | newspaper[N].unsubscribe -> CONX_NEWSPAPER_QUEUE
),
CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1:NEWS] =
(
       sendNewsAlert[d:NEWS] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1][d]
       | newspaper[N].newsAlertReceived[e1] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED
       | newspaper[N].unsubscribe -> CONX_NEWSPAPER_QUEUE
),
CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1:NEWS][e2:NEWS] =
(
       sendNewsAlert[d:NEWS] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1][e2][d]
       | newspaper[N].newsAlertReceived[e1] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e2]
       | newspaper[N].unsubscribe -> CONX_NEWSPAPER_QUEUE
),
CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1:NEWS][e2:NEWS][e3:NEWS] =
(
       newspaper[N].newsAlertReceived[e1] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e2][e3]
       | newspaper[N].unsubscribe -> CONX_NEWSPAPER_QUEUE
).

|| CONX = (forall[n:NEWSPAPERS] CONX_NEWSPAPER_QUEUE(n)).
```

La función ψ_{AN1} se define por extensión de la siguiente manera:

- $\psi_{AN1}(News\ Service) =$ NEWS_SERVICE

- $\psi_{AN1}(Announcer\ Listener)$ =CONX
- $\psi_{AN1}(Newspaper\ 1)$ = NEWSPAPER1
- $\psi_{AN1}(Newspaper\ 2)$ = NEWSPAPER2
- $\psi_{AN1}(Newspaper\ 3)$ = NEWSPAPER3

obteniendo como resultado ARQ = $BehaviorAsFSP(B_{AN1})$ con:

```
|| ARQ =
(
        NEWS_SERVICE
        || CONX
        || NEWSPAPER_1
        || NEWSPAPER_2
        || NEWSPAPER_3
).
```

El comportamiento del sistema completo se encuentra modelado mediante el proceso FSP ARQ, que consiste en la composición de los procesos correspondientes a cada uno de los conectores y componentes del sistema. Se puede apreciar que la función ψ_{AN1} cumple con las restricciones impuestas por la metodología mediante la propiedad $CommunicationIsLegal$ ya que:

- $\alpha(NEWS\ SERVICE) \cap \alpha(CONX) \neq \phi \Rightarrow NEWS\ SERVICE \overset{AD}{\leftrightarrow} CONX$
- $\alpha(NEWSPAPER\ 1) \cap \alpha(CONX) \neq \phi \Rightarrow NEWSPAPER\ 2 \overset{AD}{\leftrightarrow} CONX$
- $\alpha(NEWSPAPER\ 2) \cap \alpha(CONX) \neq \phi \Rightarrow NEWSPAPER\ 2 \overset{AD}{\leftrightarrow} CONX$
- $\alpha(NEWSPAPER\ 3) \cap \alpha(CONX) \neq \phi \Rightarrow NEWSPAPER\ 3 \overset{AD}{\leftrightarrow} CONX$

con:

- $\alpha(NEWS\ SERVICE) = \{newsRedacted[NEWS],\ sendNewsAlert[NEWS]\}$
- $\alpha(CONX) = \{\ newspaper[NEWSPAPERS].newsAlertOnScreen[NEWS],$ $newspaper[NEWSPAPERS].newsAlertReceived[NEWS],$ $newspaper[NEWSPAPERS].subscribe, newspaper[NEWSPAPERS].unsuscribe, newsRedacted[NEWS],$ $sendNewsAlert[NEWS]\}$
- $\alpha(NEWSPAPER\ 1) = \{\ newspaper[1].newsAlertOnScreen[NEWS],\ newspaper[1].newsAlertReceived[NEWS],$ $newspaper[1].subscribe, newspaper[1].unsuscribe\}$
- $\alpha(NEWSPAPER\ 2) = \{\ newspaper[2].newsAlertOnScreen[NEWS],\ newspaper[2].newsAlertReceived[NEWS],$ $newspaper[2].subscribe, newspaper[2].unsuscribe\}$
- $\alpha(NEWSPAPER\ 3) = \{\ newspaper[3].newsAlertOnScreen[NEWS],\ newspaper[3].newsAlertReceived[NEWS],$ $newspaper[3].subscribe, newspaper[3].unsuscribe\}$

En el listado anterior se utiliza una notación consiste con la de FSP. Por ejemplo, `newsRedacted[NEWS]` es un shortcut para definir el conjunto de acciones `newsRedacted[NEWS_A]` y `newsRedcated[NEWS_B]` (Nótese que se modela el sistema con una cantidad fija y finita de *Listeners* y con un número acotado de *Anuncios* posibles. Esto es resultado directo del uso de LTSA, ya que esta herramienta solo puede operar sobre modelos finitos [MK99]. En la sección "Uso de LTSA" se estudian y describen las restricciones resultantes del uso de la misma).

De esta manera, se cuenta con todas las definiciones necesarias para conocer por completo el comportamiento de la descripción propuesta para el sistema.

Presentación y formalización de las propiedades de sistema

Para poder determinar la satisfacción o no de las "propiedades interesantes" por parte del comportamiento asociado propuesto, éstas deben ser especificadas. De acuerdo a la discusión del capítulo anterior, las propiedades de una descripción se pueden particionar en dos subconjuntos: propiedades de estilos y propiedades del sistema. En el desarrollo del ejemplo se menciona una única propiedad de sistema: "Para cada noticia recibida, se debe presentar una alerta en la pantalla de la redacción que la recibe". Esta propiedad se puede formalizar mediante la propiedad `DISPLAY_CORRECT_NEWS_ONSCREEN` presentada a continuación:

```
property DISPLAY_CORRECT_NEWS_ONSCREEN =
(
        newsAlertReceived[n:NEWS] -> newsAlertOnScreen[n] -> DISPLAY_CORRECT_NEWS_ONSCREEN
).
```

Presentación y formalización de las propiedades de estilos

La arquitectura presentada hace uso del estilo *Announcer-Listener* en pos de lograr los objetivos propuestos para el sistema (i.e: la satisfacción de la propiedad enunciada anteriormente). Por lo tanto, las propiedades definidas por este estilo son parte de las "propiedades interesantes" del sistema. Es necesario, entonces, contar con una especificación de dicho estilo. En el Apéndice A – Catálogo de Estilos, se pueden encontrar todas las especificaciones realizadas y utilizadas en este trabajo. El código fuente correspondiente a la especificación de este estilo se puede encontrar en el listado `AnnouncerListener.lts` del Apéndice B.

La especificación del estilo *Announcer-Listener* define las propiedades que se listan a continuación

```
property ANNOUNCER_BEHAVIOR = (announce[DATA] -> ANNOUNCER_BEHAVIOR).

property LISTENER_BEHAVIOR =
(
        subscribe -> LISTENER_BEHAVIOR_SUBSCRIBED
),
LISTENER_BEHAVIOR_SUBSCRIBED =
(
        event[DATA] -> LISTENER_BEHAVIOR_SUBSCRIBED
        | unsubscribe -> LISTENER_BEHAVIOR
).

property CORRECT_EVENTS(L=LMIN) =
```

```
(
        listener[L].subscribe -> CORRECT_EVENTS_SUBSCRIBED
        | announce[DATA] -> CORRECT_EVENTS
),
CORRECT_EVENTS_SUBSCRIBED =
(
        listener[L].unsubscribe -> CORRECT_EVENTS
        | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[d]
),
CORRECT_EVENTS_SUBSCRIBED[e1:DATA] =
(
        listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED
        | listener[L].unsubscribe -> CORRECT_EVENTS
        | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[e1][d]
),
CORRECT_EVENTS_SUBSCRIBED[e1:DATA][e2:DATA] =
(
        listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED[e2]
        | listener[L].unsubscribe -> CORRECT_EVENTS
        | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[e1][e2][d]
),
CORRECT_EVENTS_SUBSCRIBED[e1:DATA][e2:DATA][e3:DATA] =
(
        listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED[e2][e3]
        | listener[L].unsubscribe -> CORRECT_EVENTS
).
```

Todas las propiedades definidas por el estilo son propiedades de safety. Las primeras dos predican sobre el comportamiento esperado por parte de los componentes que asuman los roles de *Announcer* y *Listeners* respectivamente, mientras que la restante predica sobre los *Eventos* que deben ser recibidos por cada *Listener* y sobre el orden de los mismos. El lenguaje del estilo consiste en las acciones que son utilizadas para definir estas propiedades:

$\alpha(Announcer\ Listener) = \{listener[LISTENERS].subscribe,\ listener[LISTENERS].unsuscribe,$
$listener[LISTENERS].event[DATA],\ announe[DATA]\}$

Cabe aclarar que la definición un estilo debe especificar los rangos y constantes apropiados para expresar las propiedades del mismo. En este caso, los rangos `LISTENERS` y `DATA`.

Validación de las "propiedades interesantes" del sistema

Contando con la formalización de las propiedades del estilo utilizado y de las propiedades de sistema, se puede proceder a verificar si el proceso `ARQ` (que es la especificación mediante FSP del comportamiento de la descripción arquitectónica) las satisface. Para las propiedades de sistema, es tan simple como realizar la siguiente composición y chequear que LTSA no arroje ninguna traza de error:

```
|| CHECKED_ARQ =
(
        ARQ
        || forall[n:NEWSPAPERS] newspaper[n]:DISPLAY_CORRECT_NEWS_ONSCREEN
).
```

Realizar esta verificación es posible ya que $\alpha(\texttt{DISPLAY\_CORRECT\_NEWS\_ONSCREEN}) \subseteq \alpha(B_{AN1})$, lo que implica que `DISPLAY_CORRECT_NEWS_ONSCREEN` $\in VP_{AN1}$

Cabe aclarar que cada vez que se verifica la validez de una propiedad para un proceso dado, también se verifica que el proceso está libre de deadlocks y que todas las acciones necesarias se ejecutan periódicamente (mediante propiedades de liveness). De esta forma se evita cometer un error muy común: pensar que la satisfacción de la propiedad es suficiente cuando, de hecho, puede ser que la propiedad se cumpla porque ninguna de las acciones implicadas ocurre jamás. Aunque no se la mencione explícitamente, toda verificación realizada a lo largo del trabajo es acompañada por la verificación de las propiedades de liveness necesarias.

Para verificar las propiedades del estilo utilizado se debe resolver un pequeño problema: el alfabeto de `ARQ` no incluye las acciones del lenguaje del estilo *Announcer-Listener,* ya que el comportamiento propuesto implementa el estilo utilizando un alfabeto propio y acorde a sus fines. En el marco de este comportamiento, un *Evento* es una noticia, un *Announcer* es una agencia de noticas y un *Listener* es un diario. Como resultado $(prop: AnnouncerListener) \notin VP_{AN1}$. Por este motivo es necesario introducir un mecanismo de traducción simple que reifique las relaciones mencionadas anteriormente.

Traducción canónica para el estilo Announcer-Listener

Una **traducción canónica** para un estilo $Style$ y un comportamiento B_{AD} asociado a una descripción AD mediante la función ψ, es una función T tal que

$$T: LTSAProcessesDescriptions \rightarrow LTSAProcessesDescriptions$$

donde

1. $Dom(T) = Img(\psi)$
2. $\alpha\big(Img(T)\big) = \alpha\big(Dom(T)\big) \cup \alpha(Style)$
3. $(T(proc) = (proc \,||translateproc) \ll \{d_1, \dots, d_n\}) \vee (T(proc) = proc)$

con:

4. $TranslationProcess_{\{s1,\dots,sm\},\{d1,\dots,dn\}}(translateproc)$
5. $\{\mathrm{d}_1, \dots, \mathrm{d}_\mathrm{n}\} \subseteq \alpha(Style)$
6. $\{\mathrm{s}_1, \dots, \mathrm{s}_\mathrm{m}\} \subseteq \alpha(proc)$
7. $\{\mathrm{d}_1, \dots, \mathrm{d}_\mathrm{n}\} \cap \alpha(proc) = \emptyset$

Dada una traducción canónica T para el estilo $Style$ se definen:

1. $BehaviorAsFSP_T(B_{AD}) \equiv \; ||_{e \,\in AD.CN \,\cup AD.CX} \; T(\,\psi(e))$

2. $TranslationProcesses(T) \equiv$

   $\{tp: LTSAProcessesDescriptions | \exists p: Dom(p), T(p) = (p \parallel tp) \ll \{a_1, \dots, a_n\}\}$

3. $DestLang(T) \equiv \cup\{\{a_1, \dots, a_n\} | \exists p: Dom(p), T(p) = (p \parallel tp) \ll \{a_1, \dots, a_n\}\}$

4. $SrcLang(T) \equiv \cup\{\alpha(p) - \{a_1, \dots, a_n\} | \exists p: Dom(p), T(p) = (p \parallel tp) \ll \{a_1, \dots, a_n\}\}$

5. $LangUsedToImplement_T(B_{AD}) \equiv \big(SrcLang(T) \cup \alpha(Style)\big) - DestLang(T)$

Nótese que la función identidad es una traducción canónica válida, que resulta en $DestLang(T) = SrcLang(T) = \emptyset$ y $LangUsedToImplement_T(S_{AD}) = \alpha(Style)$.

A partir de una traducción canónica para un estilo dado se obtiene un nuevo comportamiento para la descripción, con el cual es posible verificar la satisfacción de las propiedades del estilo. La aplicación de una traducción canónica T a un comportamiento implica la "decoración" de cada uno de los procesos de dicho comportamiento con acciones pertenecientes al conjunto $DestLang(T)$. A diferencia de los mecanismos de traducción, las traducciones canónicas "decoran" cada uno de los procesos en forma individual y no a la composición de los mismos.

Se propone la función TAL como traducción canónica para el estilo *Announcer-Listener*:

- TAL(CONX) = CONX_AS_ANNOUNCE_LISTENER_SPECIFICATION
- TAL(NEWS_SERVICE) = NEWS_SERVICE_AS_ANNOUNCE_LISTENER_SPECIFICATION
- TAL(NEWSPAPER_1) = NEWSPAPER_1_AS_ANNOUNCE_LISTENER_SPECIFICATION
- TAL(NEWSPAPER_2) = NEWSPAPER_2_AS_ANNOUNCE_LISTENER_SPECIFICATION
- TAL(NEWSPAPER_3) = NEWSPAPER_3_AS_ANNOUNCE_LISTENER_SPECIFICATION

con las siguiente definiciones:

```
CANNONICAL_STYLE_TRANSLATION_LISTENER(L=LMIN) =
(
        newspaper[L].newsAlertReceived[n:NEWS] -> listener[L].event[n] ->
        CANNONICAL_STYLE_TRANSLATION_LISTENER
        |
        newspaper[L].subscribe -> listener[L].subscribe ->
        CANNONICAL_STYLE_TRANSLATION_LISTENER
        |
        newspaper[L].unsubscribe -> listener[L].unsubscribe ->
        CANNONICAL_STYLE_TRANSLATION_LISTENER
).

CANNONICAL_STYLE_TRANSLATION_ANNOUNCE =
(
        sendNewsAlert[n:NEWS] -> announce[n] -> CANNONICAL_STYLE_TRANSLATION_ANNOUNCE
).

|| CONX_AS_ANNOUNCE_LISTENER_SPECIFICATION =
(
        CONX
        || CANNONICAL_STYLE_TRANSLATION_ANNOUNCE
        || forall[l:LISTENERS] CANNONICAL_STYLE_TRANSLATION_LISTENER(l)
)
```

```
<<
{
        listener[LISTENERS].subscribe, listener[LISTENERS].unsubscribe,
        listener[LISTENERS].event[DATA], announce[DATA]
}.

|| NEWS_SERVICE_AS_ANNOUNCE_LISTENER_SPECIFICATION  =
(
        NEWS_SERVICE || CANNONICAL_STYLE_TRANSLATION_ANNOUNCE
)
<< {announce[DATA]}.

|| NEWSPAPER_1_AS_ANNOUNCE_LISTENER_SPECIFICATION =
(
        NEWSPAPER_1 || CANNONICAL_STYLE_TRANSLATION_LISTENER(1)
)
<< {listener[1].subscribe, listener[1].unsubscribe, listener[1].event[DATA]}.

|| NEWSPAPER_2_AS_ANNOUNCE_LISTENER_SPECIFICATION =
(
        NEWSPAPER_2 || CANNONICAL_STYLE_TRANSLATION_LISTENER(2)
)
<< {listener[2].subscribe, listener[2].unsubscribe, listener[2].event[DATA]}.

|| NEWSPAPER_3_AS_ANNOUNCE_LISTENER_SPECIFICATION =
(
        NEWSPAPER_3 || CANNONICAL_STYLE_TRANSLATION_LISTENER(3)
)
<< {listener[3].subscribe, listener[3].unsubscribe, listener[3].event[DATA]}.
```

A partir de TAL se puede obtener el proceso `ARQ_AS_ANNOUNCER_LISTENER_SEPCIFICATION` = $BehaviorAsFSP_{TAL}(B_{AN1})$:

```
|| ARQ_AS_ANNOUNCE_LISTENER_SPECIFICATION =
(
        NEWS_SERVICE_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || CONX_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || NEWSPAPER_1_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || NEWSPAPER_2_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || NEWSPAPER_3_AS_ANNOUNCE_LISTENER_SPECIFICATION
).
```

De esta forma se obtiene un nuevo modelo de comportamiento para la descripción $AN1$, sobre el cual es posible verificar las propiedades del estilo *Announcer-Listener*. A partir del proceso `ARQ_AS_ANNOUNCE_LISTENER_SPECIFICATION` se puede determinar si el comportamiento B_{AN1} asociado a la descripción arquitectónica, utiliza correctamente el estilo *Announcer-Listener* al verificar la satisfacción de las propiedades del estilo:

```
|| ARQ_IS_ANNOUNCE_LISTENER =
(
        ARQ_AS_ANNOUNCE_LISTENER
        || forall[l:LISTENERS] listener[l]:LISTENER_BEHAVIOR
        || forall[l:LISTENERS] CORRECT_EVENTS(l)
        || ANNOUNCER_BEHAVIOR
).
```

En este caso particular, la composición del proceso anterior es suficiente ya que el estilo no incluye propiedades expresadas mediante LTL. Si lo hiciera, se deberían validar cada una de las mismas contra el proceso `ARQ_AS_ANNOUNCE_LISTENER`. Gracias a contar con la traducción canónica TAL,

también se verifica que el comportamiento respeta las restricciones de configuración impuestas por el estilo.

Resumen del trabajo con la descripción original

Hasta aquí se ha presentado el comportamiento propuesto para la descripción arquitectónica original y se especificaron mediante FSP las propiedades interesantes de la misma. Luego se verificó que la semántica satisface las propiedades de sistema y de los estilos utilizados. Para afirmarlo, se trabajó de acuerdo al siguiente esquema:

- **Descripción original del sistema**
  - **Proceso LTSA:**
    - `ARQ`
  - **Lenguajes exhibidos:**
    - Lenguaje original del sistema : $\alpha(B_{AN1})$
  - **Propiedades verificables (y verificadas):**
    - Originales del sistema:
      - `DISPLAY_CORRECT_NEWS_ONSCREEN`

- **Descripción del sistema como una implementación del estilo *Announcer-Listener***
  - **Proceso LTSA:**
    - ARQ_AS_ANNOUNCER_LISTENER
  - **Lenguajes exhibidos:**
    - Lenguaje original del sistema: $\alpha(B_{AN1})$
    - Lenguaje definido por el estilo *Announcer-Listener*: $\alpha(Announcer\ Listener)$
  - **Propiedades verificables (y verificadas):**
    - Originales del sistema:
      - `DISPLAY_CORRECT_NEWS_ONSCREEN`
    - Del estilo *Announcer-Listener*:
      - `ANNOUNCER_BEHAVIOR`
      - `CORRECT_EVENTS`
      - `LISTENER_BEHAVIOR`

Un posible refinamiento de la descripción original

A continuación, se presenta un comportamiento para la descripción arquitectónica que se pretende sea un refinamiento adecuado de la descripción original. Se define como $AN2$ a la descripción arquitectónica representada por el siguiente diagrama (a la que también se denominará "descripción concreta") y como $B_{AN2} =< P_{AN2}, VP_{AN2}, BP_{AN2} >$ al comportamiento asociado propuesto para la misma. Como se hiciera con la descripción más abstracta, se define la

función ψ_{AN2} al proveer un proceso FSP para cada uno de los componentes y conectores de esta nueva descripción:

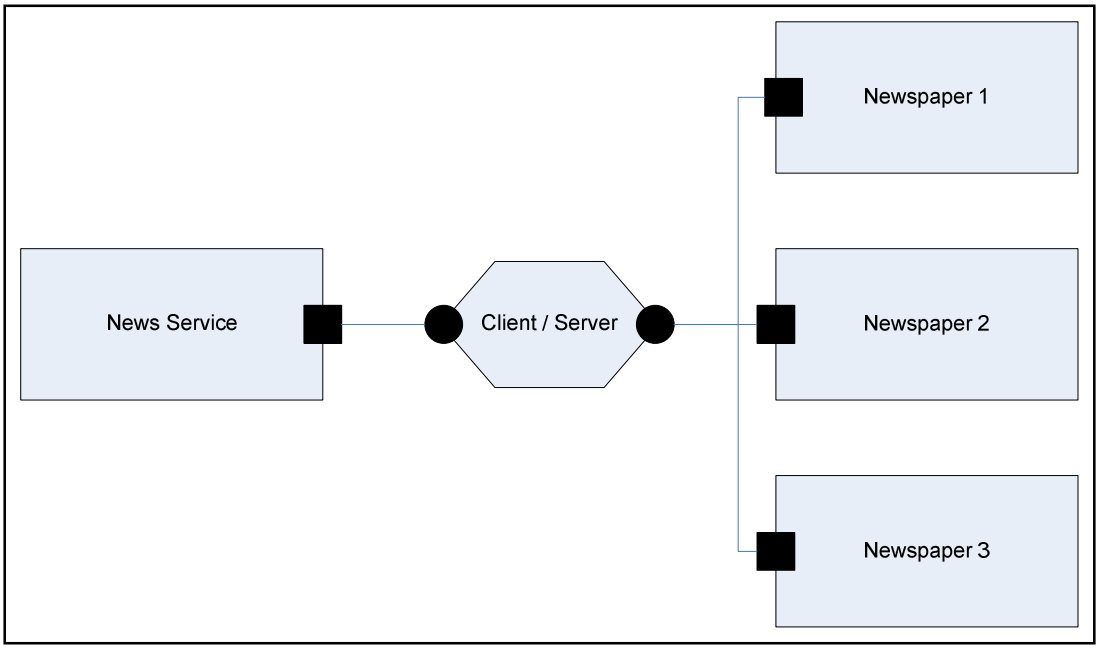

Figura 4-2: Descripción del sistema de agencia de noticias utilizando el estilo Client-Server

```
const NEWSPAPER_MIN = 1
const NEWSPAPER_MAX = 3
range NEWSPAPERS = NEWSPAPER_MIN..NEWSPAPER_MAX

const NEWS_MIN = 0
const NEWS_MAX = 1
range NEWS = NEWS_MIN..NEWS_MAX

const QUERY_NEWS_CALL = 1
const CANCEL_NEWS_SUBSCRIPTION = 2

const NO_NEWS = NEWS_MIN - 1

// CS SPECIFICATION CONSTS
// -----------------------

const CLIENT_MIN = NEWSPAPER_MIN
const CLIENT_MAX = NEWSPAPER_MAX
range CLIENTS = CLIENT_MIN..CLIENT_MAX

const CALL_MIN = QUERY_NEWS_CALL
const CALL_MAX = CANCEL_NEWS_SUBSCRIPTION
range CLIENT_CALLS = CALL_MIN..CALL_MAX

const RESPONSE_MIN = NO_NEWS
const RESPONSE_MAX = NEWS_MAX
range SERVER_RESPONSES = RESPONSE_MIN..RESPONSE_MAX

range SERVER_THREADS = 0..1
range SERVER_THREADS_AND_BUSY = -1..1
const SERVER_THREAD_BUSY = -1
```

```
// NEWSPAPER
// ---------

NEWSPAPER =
(
        call[QUERY_NEWS_CALL] ->
        (
                response[NO_NEWS] -> NEWSPAPER_SUBSCRIBED
                | response[n:NEWS] ->  newsAlertOnScreen[n] -> NEWSPAPER_SUBSCRIBED
                | dos -> NEWSPAPER
        )
),
NEWSPAPER_SUBSCRIBED =
(
        call[QUERY_NEWS_CALL] ->
        (
                response[NO_NEWS] -> NEWSPAPER_SUBSCRIBED
                | response[n:NEWS] ->  newsAlertOnScreen[n] -> NEWSPAPER_SUBSCRIBED
                | dos -> NEWSPAPER_SUBSCRIBED
        )
        | call[CANCEL_NEWS_SUBSCRIPTION] ->
        (
                response[SERVER_RESPONSES] -> NEWSPAPER
                | dos -> NEWSPAPER_SUBSCRIBED
        )
).


|| NEWSPAPER_1 = (client[1]:NEWSPAPER)
/ {newspaper[1].newsAlertOnScreen[n:NEWS] / client[1].newsAlertOnScreen[n]}.

|| NEWSPAPER_2 = (client[2]:NEWSPAPER)
/ {newspaper[2].newsAlertOnScreen[n:NEWS] / client[2].newsAlertOnScreen[n]}.

|| NEWSPAPER_3 = (client[3]:NEWSPAPER)
/ {newspaper[3].newsAlertOnScreen[n:NEWS] / client[3].newsAlertOnScreen[n]}.

// NEWS_SERVICE
// ------------

SERVER_THREAD(T=0) =
(
        st[T].call[c:CLIENTS][CANCEL_NEWS_SUBSCRIPTION] ->
        st[T].response[c][SERVER_RESPONSES] ->
        SERVER_THREAD
        |
        st[T].call[c:CLIENTS][QUERY_NEWS_CALL] ->
        st[T].response[c][SERVER_RESPONSES] ->
        SERVER_THREAD
).

SERVER_NEWS_GENERATION =
(
        newsRedacted[n:NEWS] -> enqueueNewsAlert[NEWS] -> SERVER_NEWS_GENERATION
).

SERVER_NEWS_QUEUES(C=CLIENT_MIN) =
(
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION] ->
        st[SERVER_THREADS].response[C][NO_NEWS] ->
        SERVER_NEWS_QUEUES
        |
        st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] ->
        st[SERVER_THREADS].response[C][NO_NEWS] ->
        SERVER_NEWS_QUEUES_SUBSCRIBED
        |
        enqueueNewsAlert[n:NEWS] -> SERVER_NEWS_QUEUES
),
SERVER_NEWS_QUEUES_SUBSCRIBED =
(
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION] ->
```

```
        st[SERVER_THREADS].response[C][NO_NEWS] ->
        SERVER_NEWS_QUEUES
        |
        st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] ->
        st[SERVER_THREADS].response[C][NO_NEWS] ->
        SERVER_NEWS_QUEUES_SUBSCRIBED
        |
        enqueueNewsAlert[n:NEWS] -> SERVER_NEWS_QUEUES[n]
),
SERVER_NEWS_QUEUES[n1:NEWS] =
(
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION] ->
        st[SERVER_THREADS].response[C][NO_NEWS] ->
        SERVER_NEWS_QUEUES
        |
        st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] ->
        st[SERVER_THREADS].response[C][n1] ->
        SERVER_NEWS_QUEUES_SUBSCRIBED
        |
        enqueueNewsAlert[n:NEWS] -> SERVER_NEWS_QUEUES[n1][n]
),
SERVER_NEWS_QUEUES[n1:NEWS][n2:NEWS] =
(
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION] ->
        st[SERVER_THREADS].response[C][NO_NEWS] ->
        SERVER_NEWS_QUEUES
        |
        st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] ->
        st[SERVER_THREADS].response[C][n1] ->
        SERVER_NEWS_QUEUES[n2]
).

|| NEWS_SERVICE =
(
        SERVER_NEWS_GENERATION
        || forall[t:SERVER_THREADS] SERVER_THREAD(t)
        || forall[c:CLIENTS] SERVER_NEWS_QUEUES(c)
).

// CONNECTOR
// ---------

CS_CONX_ST_STATUS(T=0) = CS_CONX_ST_STATUS_FREE,
CS_CONX_ST_STATUS_FREE =
(
        st[T].call[c:CLIENTS][CLIENT_CALLS] -> CS_CONX_ST_STATUS_BUSY[c]
),
CS_CONX_ST_STATUS_BUSY[c:CLIENTS] =
(
        st[SERVER_THREAD_BUSY].call[CLIENTS][CLIENT_CALLS] -> CS_CONX_ST_STATUS_BUSY[c]
        | st[T].response[c][SERVER_RESPONSES] -> CS_CONX_ST_STATUS_FREE
).

CS_CONX_CLIENT(C=0) =
(
        client[C].call[call:CLIENT_CALLS] ->
        (
                st[s:SERVER_THREADS].call[C][call] ->
                st[s].response[C][r:SERVER_RESPONSES] ->
                client[C].response[r]  ->
                CS_CONX_CLIENT
                |
                st[SERVER_THREAD_BUSY].call[C][call] -> CS_CONX_CLIENT
        )
).

|| CS_CONX =
(
        forall[t:SERVER_THREADS] CS_CONX_ST_STATUS(t)
        || forall[c:CLIENTS] CS_CONX_CLIENT(c)
)
```

```
/ {client[c:CLIENTS].dos / st[SERVER_THREAD_BUSY].call[c][CLIENT_CALLS]}.
```

La función ψ_{AN2} queda definida de la siguiente forma:

- $\psi_{AN2}(News\ Service) =$ NEWS_SERVICE
- $\psi_{AN2}(Client\ Server) =$CS_CONX
- $\psi_{AN2}(Newspaper\ 1) =$ NEWSPAPER1
- $\psi_{AN2}(Newspaper\ 2) =$ NEWSPAPER2
- $\psi_{AN2}(Newspaper\ 3) =$ NEWSPAPER3

y como resultado ARQREF $= BehaviorAsFSP(B_{AN2})$ con:

```
|| ARQREF =
(
        CS_CONX
        || NEWSPAPER_1
        || NEWSPAPER_2
        || NEWSPAPER_3
        || NEWS_SERVICE
).
```

En esta descripción se reemplaza el uso del estilo *Announcer-Listener* por el estilo *Client-Server,* dando como resultado un cambio en la forma en que los componentes interactúan y en sus funciones. Si esta descripción arquitectónica es un refinamiento adecuado, entonces dichos cambios no deberán influenciar su capacidad para satisfacer las propiedades interesantes de la descripción original.

Traducción canónica para el estilo Client-Server

En primer lugar se verifica que la semántica propuesta se comporta, efectivamente, como un sistema *Client-Server* (cuya especificación puede encontrarse en el Apéndice A y el código fuente correspondiente a la misma, en el listado `ClientServer.lts` del Apéndice B). Se procede en forma análoga a lo realizado con la descripción más abstracta, proveyendo la traducción canónica TCS:

- TCS(CONX) = CONX_AS_CS_SPECIFICATION
- TCS(NEWS_SERVICE) = NEWS_SERVICE_AS_CS_SPECIFICATION
- TCS(NEWSPAPER_1) = NEWSPAPER_1
- TCS(NEWSPAPER_2) = NEWSPAPER_2
- TCS(NEWSPAPER_3) = NEWSPAPER_3

con las siguientes definiciones:

```
CANNONICAL_CS_TRANSLATION_ASSOCIATE  =
(
       st[t:SERVER_THREADS].call[c:CLIENTS][call:CLIENT_CALLS] -> associate[t][c] ->
       st[t].clientCall[call] -> CANNONICAL_CS_TRANSLATION_ASSOCIATE
).

CANNONICAL_CS_TRANSLATION_DISASSOCIATE  =
(
       st[t:SERVER_THREADS].response[c:CLIENTS][response:SERVER_RESPONSES] ->
       st[t].clientResponse[response] -> disassociate[t][c] ->
       CANNONICAL_CS_TRANSLATION_DISASSOCIATE
).

|| NEWS_SERVICE_AS_CS_SPECIFICATION =
(
       NEWS_SERVICE
       || CANNONICAL_CS_TRANSLATION_ASSOCIATE
       || CANNONICAL_CS_TRANSLATION_DISASSOCIATE
)
<<
{
       associate[t:SERVER_THREADS][c:CLIENTS],
       disassociate[t:SERVER_THREADS][c:CLIENTS],
       st[SERVER_THREADS].clientCall[CLIENT_CALLS],
       st[SERVER_THREADS].clientResponse[SERVER_RESPONSES]
}.

|| CS_CONX_AS_CS_SPECIFICATION =
(
       CS_CONX
       || CANNONICAL_CS_TRANSLATION_ASSOCIATE
       || CANNONICAL_CS_TRANSLATION_DISASSOCIATE
)
<<
{
       associate[t:SERVER_THREADS][c:CLIENTS],
       disassociate[t:SERVER_THREADS][c:CLIENTS],
       st[SERVER_THREADS].clientCall[CLIENT_CALLS],
       st[SERVER_THREADS].clientResponse[SERVER_RESPONSES]
}.
```

A partir de TCS se puede obtener el proceso `ARQREF_AS_CS_SEPCIFICATION` = $BehaviorAsFSP_{TCS}(B_{AN2})$ de la siguiente forma:

```
|| ARQREF_AS_CS_SPECIFICATION =
(
       CS_CONX_AS_CS_SPECIFICATION
       || NEWSPAPER_1
       || NEWSPAPER_2
       || NEWSPAPER_3
       || NEWS_SERVICE_AS_CS_SPECIFICATION
).
```

y verificar el cumplimiento de cuatro de las cinco propiedades de safety propuestas por el estilo:

```
|| ARQREF_IS_CS =
(
       ARQREF_AS_CS_SPECIFICATION
       || forall[c:CLIENTS] client[c]:CLIENT_BEHAVIOR
       || forall[t:SERVER_THREADS] SERVER_THREAD_BEHAVIOR(t)
       || forall[t:SERVER_THREADS] SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR(t)
       || forall[c:CLIENTS] ASSOCIATION_KEEPS_CLIENT_CALL(c)
       || forall[c:CLIENTS] ASSOCIATION_KEEPS_SERVER_RESPONSE(c)
```

```
).
```

Mediante LTSA se debe verificar que el proceso `ARQREF_AS_CS_SPECIFICATION` también satisface las propiedades del estilo expresadas como formulas FLTL: `DOS_ONLY_IF_ALL_TRHEADS_ARE_BUSY` y `ASSOCIATE_NOT_BUSY_THREADS` (efectivamente lo hace). Es importante notar que la satisfacción de la quinta propiedad de safety del estilo (`FAIR_ASSOCIATIONS`) no es verificada. La omisión es adrede y responde al hecho de que dicha propiedad no es requerida para el correcto funcionamiento de la descripción arquitectónica analizada. Es decir, se eligió utilizar un sub-estilo del estilo *Client-Server* que no implica la satisfacción de dicha propiedad.

Mecanismo de traducción

A continuación, y de acuerdo a la metodología propuesta, es necesario proveer un mecanismo de traducción que permita interpretar los estilos de la descripción arquitectónica más abstracta en términos de los estilos de la descripción arquitectónica más concreta. De acuerdo a la instanciación con LTSA de la metodología, la tarea citada es equivalente a encontrar un conjunto de procesos de traducción cuyo lenguaje de origen sea $\alpha(B_{AN2}) - \alpha(B_{AN1})$ y el lenguaje de destino sea $LangUsedToImplement_{TAL}(B_{AN1})$.

El conjunto de procesos de traducción propuesto es {`REFINEMENT_TRANSLATION`, `REFINEMENT_TRANSLATION_SUBSCRIBE_S(CLIENTS)`} cuyos elementos se detallan a continuación:

```
REFINEMENT_TRANSLATION =
(
        client[c:CLIENTS].response[n:NEWS] ->
        newspaper[c].newsAlertReceived[n] ->
        REFINEMENT_TRANSLATION
        |
        st[SERVER_THREADS].call[c:CLIENTS][CANCEL_NEWS_SUBSCRIPTION] ->
        newspaper[c].unsubscribe ->
        REFINEMENT_TRANSLATION
        |
        enqueueNewsAlert[n:NEWS] ->
        sendNewsAlert[n] ->
        REFINEMENT_TRANSLATION
).

REFINEMENT_TRANSLATION_SUBSCRIBE(C=CLIENT_MIN) =
(
        st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] ->
        newspaper[C].subscribe ->
        REFINEMENT_TRANSLATION_SUBSCRIBE_S
        |
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION] ->
        REFINEMENT_TRANSLATION_SUBSCRIBE
),
REFINEMENT_TRANSLATION_SUBSCRIBE_S =
(
        st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] ->
        REFINEMENT_TRANSLATION_SUBSCRIBE_S
        |
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION] ->
        REFINEMENT_TRANSLATION_SUBSCRIBE
).
```

Los procesos REFINEMENT_TRANSLATION y REFINEMENT_TRANSLATION_SUBSCRIBE_S "traducen" del lenguaje de la descripción concreta, al lenguaje utilizado por la descripción original para implementar el estilo *Announce-Listener* ($LangUsedToImplement_{TAL}(B_{AN1})$). El primer proceso determina que:

1. una respuesta del servidor con un dato en el rango NEWS es equivalente a recibir un *Evento*
2. una llamada CANCEL_NEWS_SUBSCRIPTION recibida por el servidor es equivalente a una *Desuscripción*
3. la acción enqueueNewsAlert[n:NEWS] es equivalente al anuncio de un *Evento*.

El segundo proceso establece que la primera llamada QUERY_NEWS_CALL es equivalente a una *Suscripcion*, al igual que toda llamada QUERY_NEWS_CALL posterior a una llamada CANCEL_NEWS_SUBSCRIPTION.

Utilización del mecanismo de traducción para validar propiedades interesantes de sistema

A partir del mecanismo de traducción propuesto, se obtiene un nuevo comportamiento definido a partir de un único proceso al que se llamará ARQREF_AS_ABSTRACT_DESCRIPCTION_LANGUAGE:

```
|| ARQREF_AS_ABSTRACT_DESCRIPCTION_LANGUAGE =
(
        ARQREF
        || REFINEMENT_TRANSLATION
        || forall[c:CLIENTS] REFINEMENT_TRANSLATION_SUBSCRIBE(c)
)
<<
{
        newspaper[NEWSPAPERS].unsubscribe,
        newspaper[NEWSPAPERS].subscribe,
        newspaper[NEWSPAPERS].newsAlertReceived[n:NEWS],
        sendNewsAlert[n:NEWS]
}.
```

Con este nuevo modelo de comportamiento es posible validar la satisfacción de la propiedad DISPLAY_CORRECT_NEWS_ONSCREEN, la única propiedad de sistema especificada para la descripción original. Esto es posible ya que α(DISPLAY_CORRECT_NEWS_ONSCREEN) $\subseteq$ α(ARQREF_AS_ABSTRACT_DESCRIPCTION_LANGUAGE). Mediante la siguiente composición:

```
|| ARQREF_IS_REFINEMENT_SYSTEM_PROPS =
(
        ARQREF_AS_ABSTRACT_DESCRIPCTION_LANGUAGE
        || forall[n:NEWSPAPERS] newspaper[n]:DISPLAY_CORRECT_NEWS_ONSCREEN
).
```

se verifica la satisfacción de la propiedad DISPLAY_CORRECT_NEWS_ONSCREEN

Utilización del mecanismo de traducción para validar propiedades de estilos interesantes

Hasta aquí se ha verificado la satisfacción de la propiedad DISPLAY_CORRECT_NEWS_ONSCREEN solamente, pero no la satisfacción de las propiedades del estilo *Announcer-Listener* que también forman parte de las propiedades interesantes de la descripción original. Sin embargo, y al igual

que ocurriera con la descripción abstracta, no se pueden verificar estas propiedades dado que están expresadas en el lenguaje específico del estilo *Announcer-Listener*. A igual problema, igual solución: se utiliza la traducción canónica TAL definida para la descripción abstracta, de la siguiente forma:

```
|| ARQREF_AS_ANNOUNCER_LISTENER_SPECIFICATION =
(
        ARQREF_AS_ABSTRACT_DESCRIPCTION_LANGUAGE
        || forall[l:LISTENERS] CANNONICAL_STYLE_TRANSLATION_LISTENER(l)
        || CANNONICAL_STYLE_TRANSLATION_ANNOUNCE
)
<<
{
        listener[LISTENERS].unsubscribe,
        listener[LISTENERS].subscribe,
        listener[LISTENERS].event[DATA],
        announce[DATA]
}.
```

Es decir, se compone el proceso `ARQREF_AS_ANNOUNCER_LISTENER_SPECIFICATION` con los procesos del conjunto $TranslationProcesses(TAN)$ y se define el alfabeto $DestLang(TAN)$ como alta prioridad.

Ahora se puede verificar la satisfacción de las propiedades definidas por estilo mediante la composición del siguiente proceso:

```
|| ARQREF_IS_REFINEMENT =
(
        ARQREF_AS_ANNOUNCER_LISTENER_SPECIFICATION
        || forall[l:LISTENERS] listener[l]:LISTENER_BEHAVIOR
        || forall[l:LISTENERS] CORRECT_EVENTS(l)
        || ANNOUNCER_BEHAVIOR
        || forall[n:NEWSPAPERS] newspaper[n]:DISPLAY_CORRECT_NEWS_ONSCREEN
).
```

Dado que LTSA informa que todas las propiedades son satisfechas, se puede decir que el comportamiento propuesto para de la descripción concreta cumple con las propiedades interesantes de estilo de la descripción original.

Conclusión del ejemplo

Habiendo confirmado que el comportamiento propuesto para la descripción concreta cumple con TODAS las propiedades interesantes definidas para la descripción original, se puede afirmar que la descripción concreta es un refinamiento correcto de la descripción original, de acuerdo a la interpretación propuesta (mediante los mecanismos de traducción presentados) y a las propiedades de interés definidas. El grafico presentado a continuación, resume todos los pasos realizados en esta sección:

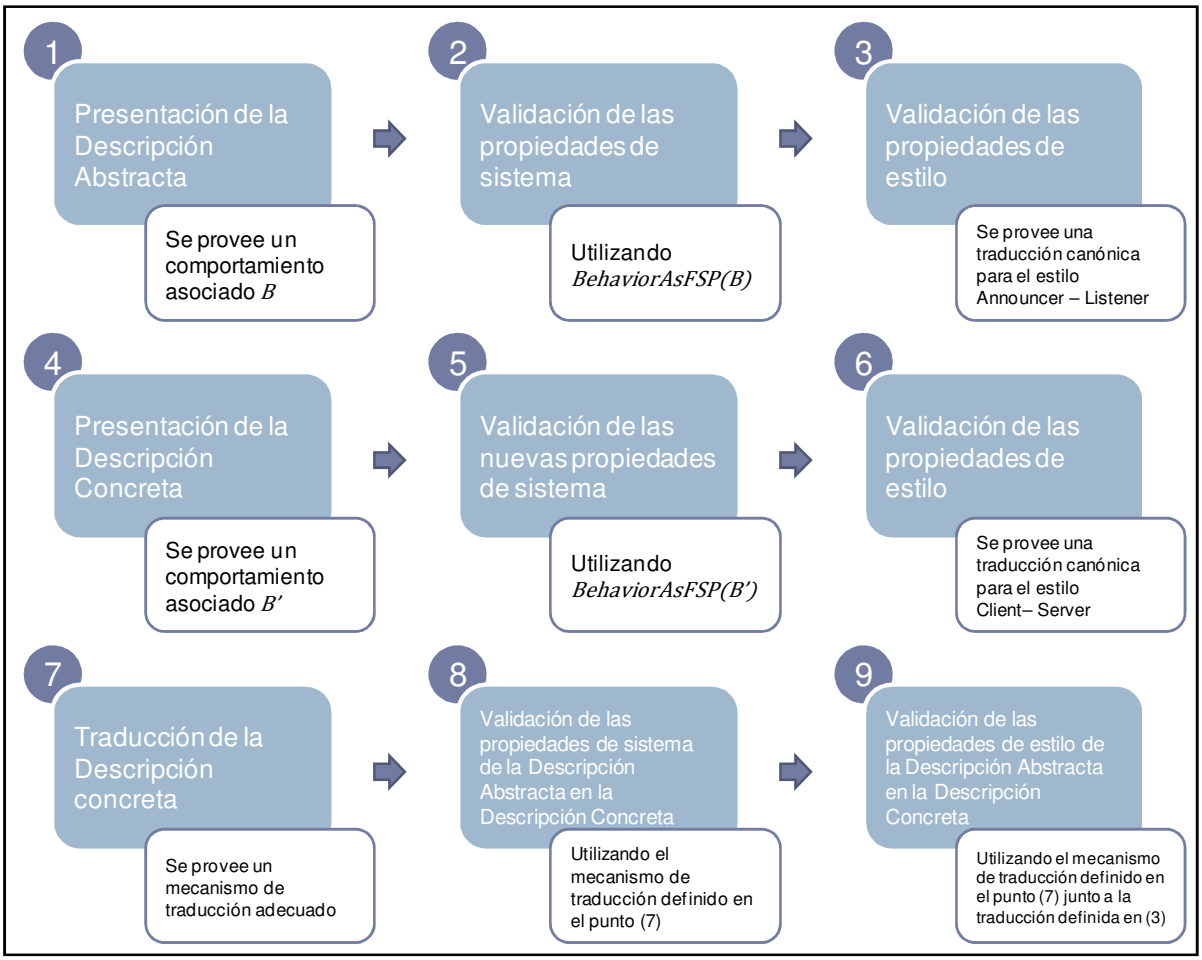

Figura 4-3: Pasos realizados para validar el refinamiento del sistema de agencia de noticias

En el gráfico que se encuentra a continuación, se resume el uso de los lenguajes y traducciones en el ejemplo. El comportamiento B_{AN1} utiliza el lenguaje $LanguageUsedToImplement_{TAL}(B_{AN1})$ para implementar el estilo *Announcer-Listener*. Mediante la traducción canónica TAL se traduce el lenguaje de B_{AN1} para validar la satisfacción de las propiedades definidas por el estilo. Por su parte, el comportamiento B_{AN2} utiliza el lenguaje $LanguageUsedToImplement_{TCS}(B_{AN2})$ para implementar el estilo *Client-Server*. Mediante la traducción canónica TCS se traduce el lenguaje de B_{AN2} para validar la satisfacción de las propiedades definidas por el estilo. Finalmente, se provee un mecanismo de traducción, que traduce del lenguaje $\alpha(B_{AN2}) - \alpha(B_{AN1})$ al lenguaje $LanguageUsedToImplement_{TAL}(B_{AN1})$.

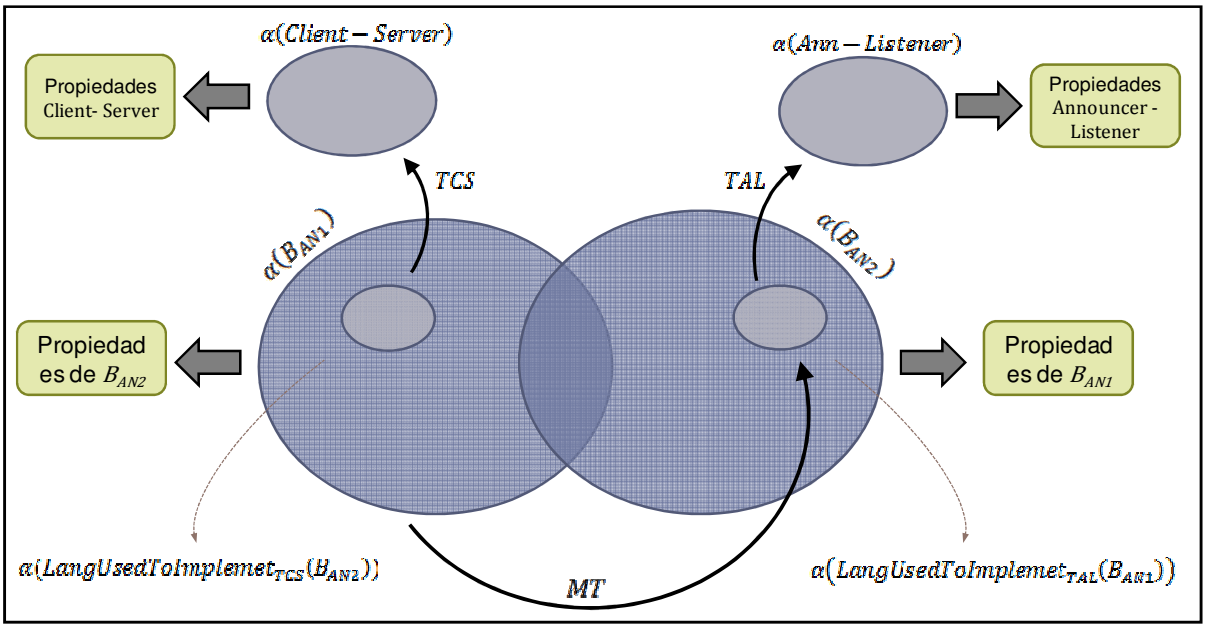

Figura 4-4: Mecanismo de traducción en relación a los diferentes lenguajes exhibidos en el ejemplo

El ejemplo presentado es sencillo, pero sirve para ilustrar la forma de trabajo propuesta por la metodología. A continuación se presenta un ejemplo más complejo en el que se muestra cómo pueden enunciarse nuevas propiedades específicas de la descripción concreta. De esta forma se sustenta la visión por la cual se identifica al concepto de refinamiento como un vehículo para recorrer en forma más seguro el gap diseño-implementación.

4.2 Un segundo ejemplo: Simulación de riesgo con Monte Carlo

El modelado y simulación de riesgo en proyectos es una técnica por la cual se puede exhibir y clasificar cuantitativamente los riesgos identificados para un proyecto dado. En la última década han aparecido varios productos que permiten realizar simulaciones de riesgo [PP6, Holte] y medir el impacto de las diferentes contingencias en un proyecto. Uno de los métodos más populares para realizar simulaciones de riesgo es la aplicación de la técnica de Monte Carlo [PMBOK]. Esta técnica se puede utilizar para calcular una distribución estadística que represente la duración total de un proyecto. Para realizarlo, dado un proyecto en el que sus tareas se encuentran relacionadas mediante relaciones de precedencia temporal (como en un diagrama Pert) y sus duraciones se encuentran especificadas mediante distribuciones estadísticas, se determina un orden topológico para las tareas y se estima, para cada una de ellas, la fecha de inicio y duración utilizando la distribución asociada (afectando la fecha de inicio de las tareas subsiguientes). Este proceso se repite varios miles de veces para obtener una distribución estadística de la duración total del proyecto[9]. En esta sección se presenta la arquitectura del componente de un sistema que realiza este tipo de simulaciones. El código fuente completo para este ejemplo se puede encontrar en los listados `MonteCarlo_PipeFilter.lts` y `MonteCarlo_ClientServer_Signal.lts` del Apéndice B.

[9] En realidad, las simulaciones son mucho más complejas e incluyen varios cálculos más que toman en cuenta la concurrencia de tareas y otros aspectos del proyecto. Se presenta una simplificación para que sea funcional al ejemplo presentado.

Resolución de Monte Carlo mediante una primera aproximación

La primera descripción arquitectónica propuesta, a la que se llamará $MonteCarlo1$, se presenta mediante el siguiente diagrama:

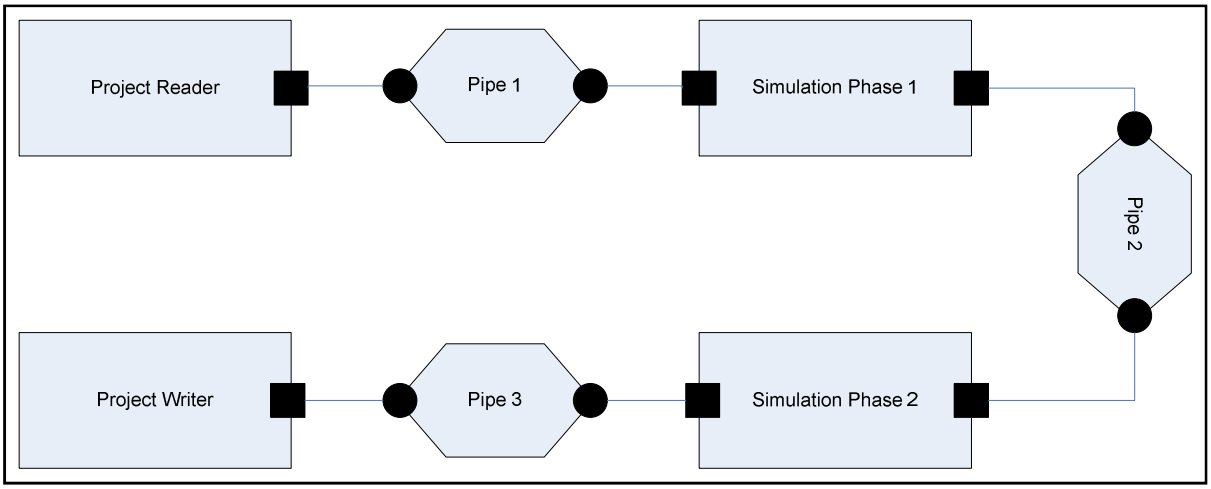

Figura 4-5: Simulación de riesgo utilizando el estilo Pipe + Filter

El componente *Project Reader* lee la información de cada una de las tareas del proyecto, los componentes *Simulation Phase 1* y *2* aplican, cada uno, una iteración del proceso de simulación de riesgo y el componente *Project Writer* escribe los resultados obtenidos. Es decir, esta descripción modela una simulación de dos iteraciones sobre un proyecto. Se puede ver que la arquitectura se corresponde con un *Pipeline* [CBB+03] con un único *Producer (Project Reader)*, un único *Sink (Project Writer)* y dos *Filtros (Simulation Phase 1* y *2)*, interconectados mediante *Pipes*. Se propone el comportamiento B_{M1} caracterizada por la función ψ_{M1} para esta descripción. ψ_{M1} se define por extensión de la siguiente forma:

- $\psi_{M1}(Project\ Writer) =$ `PROJECT_WRITER`
- $\psi_{M1}(Project\ Reader) =$ `PROJECT_READER`
- $\psi_{M1}(Simulation\ Phase\ 1) =$ `SIMULATION_PHASE_1`
- $\psi_{M1}(Simulation\ Phase\ 2) =$ `SIMULATION_PHASE_2`
- $\psi_{M1}(Pipe\ 1) =$ `p1::PIPE`
- $\psi_{M1}(Pipe\ 2) =$ `p2::PIPE`
- $\psi_{M1}(Pipe\ 3) =$ `p3::PIPE`

con el siguiente código FSP asociado:

```
const TASK_RISK_MIN = 0
const TASK_RISK_MAX = 1
range TASK_RISK = TASK_RISK_MIN..TASK_RISK_MAX
```

```
// PIPE
// ----
PIPE = EMPTY_PIPE,
EMPTY_PIPE =
(
       in[x:TASK_RISK] -> PIPE[x]
),
PIPE[d1:TASK_RISK] =
(
       in[x:TASK_RISK] -> PIPE[x][d1]
       | out[d1] -> PIPE
),
PIPE[d2:TASK_RISK][d1:TASK_RISK] =
(
       in[x:TASK_RISK] -> PIPE[x][d2][d1]
       | out[d1] -> PIPE[d2]
),
PIPE[d3:TASK_RISK][d2:TASK_RISK][d1:TASK_RISK] =
(
       out[d1] -> PIPE[d3][d2]
).

// PROJECT READER
// --------------
PROJECT_READER = (readTask[d:TASK_RISK] -> in[d] -> PROJECT_READER).

// PROJECT WRITER
// --------------
PROJECT_WRITER = (out[d:TASK_RISK] -> writeTaskRisk[d] -> PROJECT_WRITER).

// SIMULATION PHASES
// -----------------
SIMULATION_PHASE =
(
       out[d:DATA] -> if (d == TASK_RISK_MAX) then
                      (in[TASK_RISK_MIN] -> SIMULATION_PHASE)
              else
                      (in[d + 1] -> SIMULATION_PHASE)
).

|| SIMULATION_PHASE_1 = SIMULATION_PHASE
       / { pipe[1].out[d:TASK_RISK] / out[d], pipe[2].in[d:TASK_RISK] / in[d]}.

|| SIMULATION_PHASE_2 = SIMULATION_PHASE
       / { pipe[2].out[d:TASK_RISK] / out[d], pipe[3].in[d:TASK_RISK] / in[d]}.
```

como resultado ARQ $= BehaviorAsFSP(B_{M1})$ con:

```
// ARCHITECTURE
// ------------
|| ARQ =
(
       PROJECT_READER / { pipe[1].in[d:TASK_RISK] / in[d]}
       || PROJECT_WRITER / { pipe[3].out[d:TASK_RISK] / out[d]}
       || SIMULATION_PHASE_1
       || SIMULATION_PHASE_2
       || pipe[1]::PIPE
       || pipe[2]::PIPE
       || pipe[3]::PIPE
).
```

El comportamiento del sistema se especifica mediante las siguientes tres propiedades de sistema:

```
property SIMULATION_PHASE_WORK =
(
       out[d:TASK_RISK] -> if(d == TASK_RISK_MAX) then
                           (in[TASK_RISK_MIN] -> SIMULATION_PHASE_WORK)
```

```
                        else
                                (in[d + 1]  -> SIMULATION_PHASE_WORK)
).

PUSH_PROJECT_TO_PIPELINE =
(
        readTask[d:TASK_RISK] -> pipe[1].in[d] -> PUSH_PROJECT_TO_PIPELINE
).

PULL_PROJECT_FROM_PIPELINE =
(
        pipe[3].out[d:TASK_RISK] -> writeTaskRisk[d] -> PULL_PROJECT_FROM_PIPELINE
).
```

La primera propiedad especifica el comportamiento esperado para cada una de las iteraciones de simulación, o dicho de otra forma, especifica el comportamiento de los *Filtros*. La segunda propiedad especifica el comportamiento del *Producer,* al enunciar que toda tarea leída debe ser ingresada al pipeline y la última hace lo propio con el *Sink,* al enunciar que toda información extraída del pipeline debe ser persistida. El comportamiento propuesto satisface todas las propiedades presentadas.

Uso correcto del estilo Pipe + Filter

Para verificar que el comportamiento propuesta hace honor a la especificación del estilo *Pipe + Filter* (que puede encontrarse como parte del catálogo de estilos del Apéndice A y el código fuente correspondiente en el listado `PipeFilter.lts` del Apéndice B) no es necesario proveer ninguna traducción canónica (es decir, la traducción canónica a utilizar se encuentra caracterizada por la función identidad) ya que los procesos definidos por el comportamiento utilizan el mismo lenguaje que la especificación. Mediante el siguiente código FSP se puede verificar que:

```
|| CHECKED_ARQ =
(
        ARQ
        || pipe[1]::PIPE_BEHAVIOR
        || pipe[2]::PIPE_BEHAVIOR
        || pipe[3]::PIPE_BEHAVIOR
        || PRODUCER_BEHAVIOR / { pipe[1].in[d:DATA] / pipeOut.in[d]}
        || SINK_BEHAVIOR / { pipe[3].out[d:DATA] / pipeIn.out[d]}
        || FILTER_1_1
                / { pipe[1].out[d:DATA] / pipeIn.out[d], pipe[2].in[d:DATA] / pipeOut.in[d]}
        || FILTER_1_1
                / { pipe[2].out[d:DATA] / pipeIn.out[d], pipe[3].in[d:DATA] / pipeOut.in[d]}
        || FILTER_BUFFER_1_1(0,6)
                / { pipe[1].out[d:DATA] / pipeIn.out[d], pipe[3].in[d:DATA] / pipeOut.in[d]}
}
```

1. Los tres *Pipes* se comportan de acuerdo a lo especificado por el estilo

2. Los componentes *Project Writer* y *Project Reader* se comportan como un *Producer* y un *Sink* respectivamente

3. Ambos *Filtros* (*Simulation Phase Work 1* y *2*) son filtros 1-1 (por cada elemento leído como entrada por el *Filtro* un único elemento es escrito como salida, sin buffering de elementos)

4. Al conjunto formado por los componentes *Simulation Phase Work 1* y *2* el conector *Pipe 2*, se los puede considerar como un *Filtro* compuesto 1-1 con 6 slots de buffering

De especial importancia es la última propiedad, ya que la abstracción de secciones del pipeline como filtros compuestos es una de las características distintivas del estilo. La descripción presentada se puede abstraer, entonces, como se muestra en el siguiente diagrama:

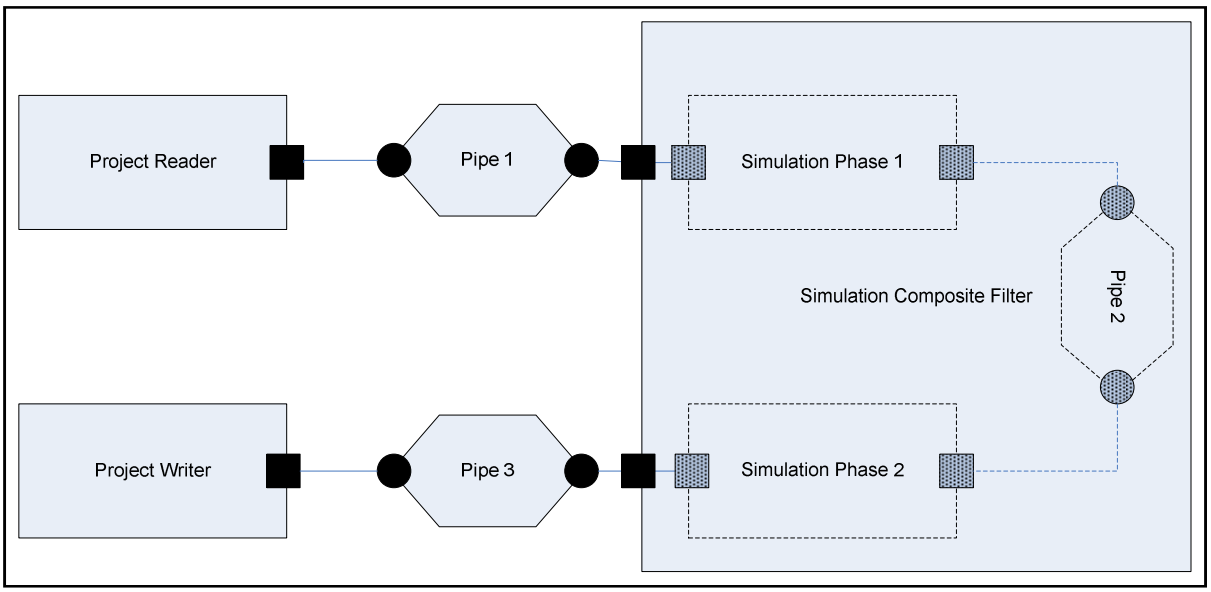

Figura 4-6: Composición de secciones del pipeline

Si se ha observado con atención el código FSP del proceso `CHECKED_ARQ`, el lector habrá podido notar el uso de los operadores de renombre y prefijo aplicados a los procesos correspondientes a las propiedades del estilo. Esto se debe al hecho de que la arquitectura incluye varios *Pipes* y varios *Filters* diferentes, pero el estilo define las propiedades para un único *Pipe* y un único *Filter.* Como resultado, la misma propiedad debe aplicarse a varias instanciaciones de los componentes y conectores definidos por el estilo (cada uno exhibiendo un lenguaje propio al que debe amoldarse la propiedad). El uso de estos mecanismos, que responde a la reutilización del código FSP que define las propiedades, no debe confundirse con el concepto de traducción canónica que es una forma de reificar la adaptación de un lenguaje a otro. Este tema se profundiza más adelante en la sección “Uso de LTSA”.

Refinar para proveer mas detalles

Modelar el proceso de simulación como un *Pipeline* supone varias ventajas: Los roles están bien definidos, se puede comprender el proceso como una composición de funciones (definiendo como consecuencia, una teoría de base formal para el sistema) y existe la posibilidad de aplicar herramientas y análisis creados especialmente para el estilo [SW99] en aras de validar los resultados obtenidos. Sin embargo, existen algunos aspectos prácticos del sistema cuya implementación se dificulta al modelarlo como un *Pipeline*. En particular, dado que un proceso de

simulación puede llevar varios minutos u horas de procesamiento, es deseable contar con la posibilidad de visualizar resultados intermedios del proceso. Si bien éstos no proveen la misma certeza que los resultados finales, son un indicativo de lo que se puede esperar.

Obtener resultados intermedios mediante la arquitectura propuesta es difícil, ya que se debería interrogar en forma individual a cada uno de los *Filtros* (que si bien en este ejemplo son solo dos, podrían llegar a ser varios miles, dependiendo de la precisión requerida). No existe un repositorio común de datos, los resultados intermedios se encuentran dispersos en las diferentes etapas del *Pipeline*. Esta propiedad es un resultado directo del uso del estilo *Pipe + Filter* para modelar el sistema. Se propone refinar el sistema, cambiando el estilo utilizado por otros que permitan describir el comportamiento propuesto más fácilmente, pero conservando las propiedades interesantes del modelo original. El resultado es una nueva descripción, a la que se llamará $MonteCarlo2$, descripta mediante el siguiente diagrama:

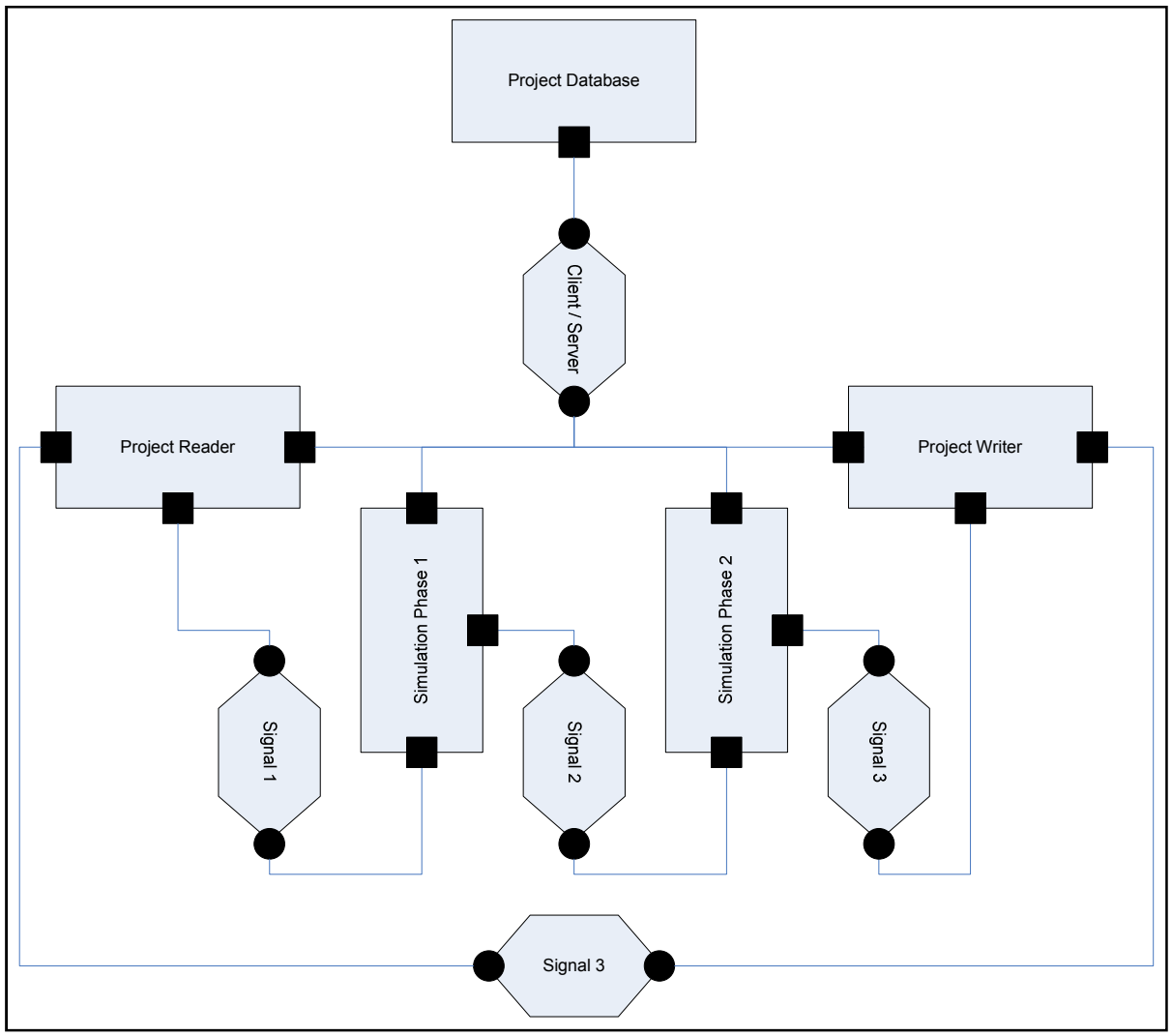

Figura 4-7: Simulación de riesgo utilizando los estilos Client-Server y Signal

A diferencia del ejemplo anterior, donde el refinamiento consistía en el cambio de un único conector, aquí se modifica completamente la configuración de la arquitectura al incluir un nuevo componente y al cambiar todos los conectores existentes. Se determina así, un cambio sustancial en el comportamiento exhibido. Se propone el comportamiento B_{M2}, caracterizada por la función ψ_{M2}, para la descripción presentada:

- $\psi_{M2}(Project\ Reader) =$ `PROJECT_WORK_UNIT(1,SLOT_COUNT,4,READ_PROJECT)`
- $\psi_{M2}(Simulation\ Phase\ 1) =$ `PROJECT_WORK_UNIT(2,0,2,WRITE_SUM)`
- $\psi_{M2}(Simulation\ Phase\ 2) =$ `PROJECT_WORK_UNIT(3,0,3,WRITE_SUM)`
- $\psi_{M2}(Project\ Writer) =$ `PROJECT_WORK_UNIT(4,0,1,WRITE_PROJECT)`
- $\psi_{M2}(Project\ Database) =$ `PROJECT_DATABASE`
- $\psi_{M2}(Client/Server) =$ `CS_CONX`
- $(\forall n: 1..4 \mid \psi_{M2}(Signal\ n) =$ `s[n]:SIGNAL_CONX)`

con:

```
const TASK_RISK_MIN = 1
const TASK_RISK_MAX = 2
range TASK_RISK = TASK_RISK_MIN..TASK_RISK_MAX

const SLOT_MIN = 0
const SLOT_MAX = 2
const SLOT_COUNT = (SLOT_MAX - SLOT_MIN + 1)
range SLOTS_COUNT = 0..SLOT_COUNT
range SLOTS = SLOT_MIN..SLOT_MAX

const INITIAL_DATA_VALUE = TASK_RISK_MIN

const RW_MIN = 1
const RW_MAX = 4
range RWS = RW_MIN..RW_MAX

const READ_PROJECT = 0
const WRITE_SUM = 2
const WRITE_PROJECT = 4

const CALL_READ = TASK_RISK_MIN - 1
range CALL_WRITE = TASK_RISK_MIN..TASK_RISK_MAX

const RESPONSE_OK = TASK_RISK_MIN - 1
const SERVER_RESPONSE_OK = TASK_RISK_MIN - 1
range SERVER_RESPONSES_DATA = TASK_RISK_MIN..TASK_RISK_MAX

const CLIENT_MIN = RW_MIN
const CLIENT_MAX = RW_MAX
range CLIENT_CALLS_WITH_SLOT_ARG = (TASK_RISK_MIN - 1)..TASK_RISK_MAX
const CLIENT_CALLS_WITH_SLOT_ARG_COUNT = TASK_RISK_MAX - (TASK_RISK_MIN -1) + 1

const SERVER_THREAD_MIN = 1
const SERVER_THREAD_MAX = 2

// CS SPECIFICATION CONSTS
// -----------------------
range CLIENTS = CLIENT_MIN..CLIENT_MAX
range CLIENT_CALLS = 0..(CLIENT_CALLS_WITH_SLOT_ARG_COUNT * SLOT_COUNT - 1)
range SERVER_RESPONSES = (TASK_RISK_MIN - 1)..TASK_RISK_MAX
```

```
const SERVER_THREAD_BUSY = SERVER_THREAD_MIN - 1
range SERVER_THREADS = SERVER_THREAD_MIN..SERVER_THREAD_MAX
range SERVER_THREADS_AND_BUSY = SERVER_THREAD_BUSY..SERVER_THREAD_MAX

// SIGNAL CONX
// -----------
SIGNAL_CONX = (signal -> SIGNAL_CONX).

// CS CONNECTOR
// ------------
CS_CONX_ST_STATUS(T=0) = CS_CONX_ST_STATUS_FREE,
CS_CONX_ST_STATUS_FREE =
(
       pdb[T].clientCall[c:CLIENTS][SLOTS][CLIENT_CALLS_WITH_SLOT_ARG] ->
       CS_CONX_ST_STATUS_BUSY[c]
),
CS_CONX_ST_STATUS_BUSY[c:CLIENTS] =
(
       pdb[SERVER_THREAD_BUSY].clientCall[CLIENTS][SLOTS][CLIENT_CALLS_WITH_SLOT_ARG] ->
       CS_CONX_ST_STATUS_BUSY[c]
       |
       pdb[T].clientResponse[c][SERVER_RESPONSES] ->
       CS_CONX_ST_STATUS_FREE
).

CS_CONX_CLIENT(C=0) =
(
       pwu[C].call[s:SLOTS][call:CLIENT_CALLS_WITH_SLOT_ARG] ->
       (
              pdb[st:SERVER_THREADS].clientCall[C][s][call] ->
              pdb[st].clientResponse[C][r:SERVER_RESPONSES] ->
              pwu[C].response[r] ->
              CS_CONX_CLIENT
              |
              pdb[SERVER_THREAD_BUSY].clientCall[C][s][call] ->
              CS_CONX_CLIENT
       )
).

|| CS_CONX =
(
       forall[t:SERVER_THREADS] CS_CONX_ST_STATUS(t) ||
       forall[c:CLIENTS] CS_CONX_CLIENT(c)
)
/
{
       pwu[c:CLIENTS].dos /
       pdb[SERVER_THREAD_BUSY].clientCall[c][SLOTS][CLIENT_CALLS_WITH_SLOT_ARG]
}.

// PROJECT WORK UNIT
// -----------------
CLIENT_RW(BEHAVIOR = READ_PROJECT) =
(
       when (BEHAVIOR != READ_PROJECT) call[SLOTS][CALL_READ] ->
       (
              dos -> CLIENT_RW
              | response[SERVER_RESPONSES_DATA] -> CLIENT_RW
       )

       | when (BEHAVIOR != WRITE_PROJECT) call[SLOTS][CALL_WRITE] ->
       (
              dos -> CLIENT_RW
              | response[SERVER_RESPONSE_OK] -> CLIENT_RW
       )
)
+ {call[SLOTS][CALL_READ],call[SLOTS][CALL_WRITE]}.

UNIT_RW(U=RW_MIN,P=0,BEHAVIOR = READ_PROJECT) = UNIT_RW_READ[SLOT_MIN],
UNIT_RW_READ[nextSlot:SLOTS] =
```

```
(
        when(BEHAVIOR != READ_PROJECT) pwu[U].call[nextSlot][CALL_READ] ->
        (
                pwu[U].dos -> UNIT_RW_READ[nextSlot]
                | pwu[U].response[d:SERVER_RESPONSES_DATA] -> UNIT_RW_WRITE[nextSlot][d]

        )
        |
        when (BEHAVIOR == READ_PROJECT) readTask[d:TASK_RISK] -> UNIT_RW_WRITE[nextSlot][d]
),
UNIT_RW_WRITE[nextSlot:SLOTS][d:TASK_RISK] =
(
        when (BEHAVIOR != READ_PROJECT)
                pwu[U].call[nextSlot][(TASK_RISK_MAX - d + 1)] ->
                UNIT_RW_WRITE_RESPONSE[nextSlot][d]
        | when (BEHAVIOR == READ_PROJECT)
                pwu[U].call[nextSlot][d] ->
                UNIT_RW_WRITE_RESPONSE[nextSlot][d]
        | when (BEHAVIOR == WRITE_PROJECT)
                writeTaskRisk[d] -> s[U].signal ->
                UNIT_RW_READ[(nextSlot + 1) % SLOT_COUNT]
),
UNIT_RW_WRITE_RESPONSE[nextSlot:SLOTS][d:TASK_RISK] =
(
        pwu[U].dos -> UNIT_RW_WRITE[nextSlot][d]
        |
        pwu[U].response[SERVER_RESPONSE_OK] ->
        s[U].signal ->
        UNIT_RW_READ[(nextSlot + 1) % SLOT_COUNT]
).

UNIT_PERMISSION_COUNT(U=RW_MIN,P=0,IN_SIGNAL=RW_MAX,BEHAVIOR = READ_PROJECT) =
UNIT_PERMISSION_COUNT[P],
UNIT_PERMISSION_COUNT[permissionCount:SLOTS_COUNT] =
(
        when (permissionCount > 0 && BEHAVIOR == READ_PROJECT)
                readTask[d:TASK_RISK] ->
                UNIT_PERMISSION_COUNT[permissionCount]
        | when (permissionCount > 0 && BEHAVIOR != READ_PROJECT)
                pwu[U].call[SLOTS][CALL_READ] ->
                UNIT_PERMISSION_COUNT[permissionCount]
        | when (permissionCount < SLOT_COUNT)
                s[IN_SIGNAL].signal ->
                UNIT_PERMISSION_COUNT[permissionCount + 1]
        | when (permissionCount > 0)
                s[U].signal ->
                UNIT_PERMISSION_COUNT[permissionCount - 1]
).

|| PROJECT_WORK_UNIT(U=RW_MIN,P=0,IN_SIGNAL=RW_MAX,BEHAVIOR=WRITE_PROJECT) =
(
        UNIT_RW(U,P,BEHAVIOR)
        || UNIT_PERMISSION_COUNT(U,P,IN_SIGNAL,BEHAVIOR)
        || pwu[U]:CLIENT_RW(BEHAVIOR)
).

// PROJECT DATABASE
// ----------------
CS_SERVER_THREAD(T=SERVER_THREAD_MIN) =
(
        pdb[T].clientCall[c:CLIENTS][s:SLOTS][CALL_READ] ->
        pdb[T].clientResponse[c][s][SERVER_RESPONSES_DATA] ->
        CS_SERVER_THREAD
        |
        pdb[T].clientCall[c:CLIENTS][s:SLOTS][w:CALL_WRITE] ->
        pdb[T].clientWriteResponse[c][s][SERVER_RESPONSE_OK] ->
        CS_SERVER_THREAD
).

CS_WRITE_LOCK =
(
```

```
        pdb[t:SERVER_THREADS].clientCall[c:CLIENTS][s:SLOTS][w:CALL_WRITE] ->
        updateResultPreviewData[s][w] ->
        pdb[t].clientWriteResponse[c][s][SERVER_RESPONSE_OK] ->
        CS_WRITE_LOCK
).

TASK_DATA_STORE(S=SLOT_MIN) = TASK_DATA_STORE[INITIAL_DATA_VALUE],
TASK_DATA_STORE[d:TASK_RISK] =
(
        pdb[SERVER_THREADS].clientCall[CLIENTS][S][w:CALL_WRITE] -> TASK_DATA_STORE[w]
        | pdb[SERVER_THREADS].clientResponse[CLIENTS][S][d] -> TASK_DATA_STORE[d]
).

|| PROJECT_DATABASE_A =
(
        forall[t:SERVER_THREADS] CS_SERVER_THREAD(t)
        || forall[s:SLOTS] TASK_DATA_STORE(s)
        || CS_WRITE_LOCK
).

|| PROJECT_DATABASE = (PROJECT_DATABASE_A)
/
{
        pdb[t:SERVER_THREADS].clientResponse[c:CLIENTS][s:SERVER_RESPONSES] /
                pdb[t].clientWriteResponse[c][SLOTS][s],
        pdb[t:SERVER_THREADS].clientResponse[c:CLIENTS][s:SERVER_RESPONSES] /
                pdb[t].clientResponse[c][SLOTS][s]
}.
```

obteniendo como resultado ARQREF $= BehaviorAsFSP(B_{M2})$:

```
|| ARQREF =
(
        PROJECT_DATABASE
        || PROJECT_WORK_UNIT(1,SLOT_COUNT,4,READ_PROJECT)
        || PROJECT_WORK_UNIT(2,0,1,WRITE_SUM)
        || PROJECT_WORK_UNIT(3,0,2,WRITE_SUM)
        || PROJECT_WORK_UNIT(4,0,3,WRITE_PROJECT)
        || CS_CONX
        || forall[rw:RWS] s[rw]:SIGNAL_CONX
).
```

La descripción arquitectónica $MonteCarlo2$ hace uso de los estilos *Client-Server* y *Signal* para implementar el cálculo de riesgo (La especificación de sendos estilos puede encontrarse en el Apéndice A – "Catálogo de estilos"). El componente *Project Database* hace las veces de repositorio compartido de datos con una cantidad determinada de "slots" para guardar información y es accedido por el resto de los componentes mediante un conector *Client-Server*. Todos los componentes, menos *Project Database,* se comunican entre sí mediante *Señales* para determinar el orden de acceso. Como resultado, los datos son procesados de la siguiente forma:

1. El componente *Project Reader* obtiene en forma progresiva los datos de las tareas de un proyecto y los escribe en orden en los slots 1 a $\#(Slots)$. Luego de cada escritura se avisa mediante una *Señal* al componente *Simulation Phase 1.*

2. El componente *Simulation Phase 1* lee el dato del slot n del repositorio, lo procesa y actualiza el resultado en el repositorio. Se envía un aviso mediante una *Señal* al

componente *Simulation Phase 2.* Se repite el comportamiento para cada *Señal* recibida, utilizando el slot $(n + 1 \ mod \ \#(Slots))$.

3. El componente *Simulation Phase 2* lee el dato del slot n del repositorio, lo procesa y actualiza el resultado en el repositorio. Se envía un aviso mediante una *Señal* al componente *Project Writer.* Se repite el comportamiento para cada *Señal* recibida, utilizando el slot $(n + 1 \ mod \ \#(Slots))$.

4. El componente *Project Writer* lee el dato del primer slot del repositorio y lo persiste. Se avisa mediante una *Señal* al componente *Project Reader.* Ante esta señal, el *Project Reader* determina que el slot n puede ser reutilizado para almacenar nueva información. Se repite el comportamiento para cada *Señal* recibida, utilizando el slot $(n + 1 \ mod \ \#(Slots))$.

La intención detrás de éste comportamiento es "imitar" las fases del *Pipeline* de la descripción arquitectónica $MonteCarlo1$. Mediante traducciones canónicas adecuadas, se verifica que los estilos utilizados se implementaron correctamente (El código FSP completo del ejemplo, incluyendo las traducciones canónicas, se puede encontrar en el Apéndice B de este trabajo).

Suponiendo que el componente *Project Database* disponga de tantos "slots" para guardar información como tareas haya en el proyecto, los resultados más recientes de la simulación se podrían consultar allí mismo (Si la cantidad de slots fuera menor, basta con incluir un nuevo componente que lea los resultados de *Project Database* y los copie progresivamente en un espacio de memoria destinado a tal fin. Se elije el primer escenario en aras de no complicar innecesariamente el ejemplo). Se puede enunciar la siguiente propiedad para garantizar la actualización de la información referente a los resultados intermedios cada vez que se escriba un nuevo dato en el repositorio (para luego poder visualizar convenientemente los resultados actualizados):

```
property RESULTS_PREVIEW_UPDATE =
(
        pdb[SERVER_THREADS].clientCall[CLIENTS][s:SLOTS][c:CALL_WRITE] ->
        updateResultPreviewData[s][c] ->
        RESULTS_PREVIEW_UPDATE
).
```

La satisfacción de esta propiedad, que es exclusiva de la descripción arquitectónica $MonteCarlo2$, es la razón por la cual se refinó: proveer un grado mayor de detalle para ciertos aspectos de la arquitectura que no pudieron ser descriptos adecuadamente mediante un nivel de abstracción mayor (Es decir, mediante la descripción original). El nuevo comportamiento hace honor a su "razón de ser", ya que satisface la propiedad `RESULTS_PREVIEW_UPDATE`.

El comportamiento B_{M2} se presenta como un potencial refinamiento de B_{M1}. De serlo, B_{M2} se podría seguir considerando, mediante una interpretación adecuada, una implementación del estilo *Pipe + Filter*. Dado que en el marco de la metodología propuesta una "interpretación adecuada" es un mecanismo de traducción, se propone utilizar el siguiente conjunto de procesos

de traducción {∀ n: 1..4 | TRANSLATE_IN(n, n)} ∪ {∀ n: 1..3 | TRANSLATE_OUT(n + 1, n)}, cuyos elementos se definen a continuación:

```
const PIPE_MIN = 1
const PIPE_MAX = 3
const PIPE_COUNT = PIPE_MAX - PIPE_MIN + 1
range PIPES = PIPE_MIN..PIPE_MAX

TRANSLATE_IN(U=RW_MIN,PIPE=0) =
(
        pwu[U].call[SLOTS][w:CALL_WRITE] ->
        (
                pwu[U].call[SLOTS][CALL_WRITE] -> TRANSLATE_IN
                | pwu[U].dos -> TRANSLATE_IN
                | pwu[U].response[SERVER_RESPONSE_OK] -> pipe[PIPE].in[w] -> TRANSLATE_IN
        )
        | pwu[U].dos -> TRANSLATE_IN
        | pwu[U].response[SERVER_RESPONSE_OK] -> TRANSLATE_IN
).

TRANSLATE_OUT(U=RW_MIN,PIPE=0) =
(
        pwu[U].call[SLOTS][CALL_READ] ->
        (
                pwu[U].call[SLOTS][CALL_READ] -> TRANSLATE_OUT
                | pwu[U].dos -> TRANSLATE_OUT
                | pwu[U].response[d:SERVER_RESPONSES_DATA] ->
                pipe[PIPE].out[d] ->
                TRANSLATE_OUT
        )
        | pwu[U].dos -> TRANSLATE_OUT
        | pwu[U].response[SERVER_RESPONSES_DATA] -> TRANSLATE_OUT
).
```

El proceso de traducción TRANSLATE_IN predica que cada par *llamada* CALL_WRITE – *respuesta* SERVER_RESPONSE_OK se puede considerar una escritura a un pipe. Por su parte, el proceso de traducción TRANSLATE_OUT predica que cada para *llamada* CALL_READ – *respuesta* SERVER_RESPONSE_OK se puede considerar una lectura de un pipe.

De esta forma se obtiene un mecanismo de traducción apropiado, que aplicado al comportamiento B_{M2} resulta en un nuevo comportamiento caracterizado por el proceso ARQ_AS_PIPELINE:

```
|| ARQ_AS_PIPELINE =
(
        ARQREF

        || TRANSLATE_IN(1,1)
        || TRANSLATE_OUT(2,1)
        || TRANSLATE_IN(2,2)
        || TRANSLATE_OUT(3,2)
        || TRANSLATE_IN(3,3)
        || TRANSLATE_OUT(4,3)
) <<
{
        pipe[PIPES].in[DATA],
        pipe[PIPES].out[DATA]
}.
```

El nuevo comportamiento satisface todas las propiedades interesantes (de estilo y de sistema) definidas para B_{M1}, como se puede demostrar mediante el análisis del siguiente proceso en LTSA:

```
|| ARQ_IS_REFINEMENT =
(
        ARQ_AS_PIPELINE
        || pipe[1]::PIPE_BEHAVIOR
        || pipe[2]::PIPE_BEHAVIOR
        || pipe[3]::PIPE_BEHAVIOR
        || PRODUCER_BEHAVIOR
                / { pipe[1].in[d:DATA] / pipeOut.in[d]}
        || SINK_BEHAVIOR
                / { pipe[3].out[d:DATA] / pipeIn.out[d]}
        || FILTER_1_1
                / { pipe[1].out[d:DATA] / pipeIn.out[d], pipe[2].in[d:DATA] / pipeOut.in[d]}
        || FILTER_1_1
                / { pipe[2].out[d:DATA] / pipeIn.out[d], pipe[3].in[d:DATA] / pipeOut.in[d]}
        || FILTER_BUFFER_1_1(0,6)
                / { pipe[1].out[d:DATA] / pipeIn.out[d], pipe[3].in[d:DATA] / pipeOut.in[d]}
        || SIMULATION_PHASE_WORK
                / { pipe[1].out[d:TASK_RISK] / out[d], pipe[2].in[d:TASK_RISK] / in[d]}
        || SIMULATION_PHASE_WORK
                / { pipe[2].out[d:TASK_RISK] / out[d], pipe[3].in[d:TASK_RISK] / in[d]}
        || PUSH_PROJECT_TO_PIPELINE
        || PULL_PROJECT_FROM_PIPELINE
).
```

Se verifica entonces que el comportamiento B_{M2} es un buen refinamiento para el comportamiento B_{M1} de acuerdo a la traducción propuesta, a las propiedades interesantes definidas para B_{M1} y a los motivos por los cuales se decidió refinar (obtener resultados intermedios).

Mediante el ejemplo de esta sección se intentan demostrar las ventajas de trabajar considerando múltiples niveles de abstracción de un sistema en forma conjunta. La posibilidad de componer los dos filtros en uno solo sería una cualidad muy difícil de visualizar en la descripción más concreta. La posibilidad de contar con un repositorio de datos centralizado sería una cualidad my difícil de describir mediante la descripción más abstracta. Sin embargo, ambas propiedades se expresan y exhiben claramente en el nivel de abstracción adecuado. La relación de refinamiento existente entre las descripciones provee la trazabilidad necesaria para garantizar el cumplimiento de las propiedades cuando el nivel de abstracción es el adecuado para las mismas. Se espera que el lector pueda apreciar e imaginar las posibilidades que esta forma de trabajo puede aportar al desarrollo de sistemas complejos.

4.3 Refinamientos malos y "malvados"

En las secciones anteriores de este capítulo se han presentado dos ejemplos de refinamientos correctos, de acuerdo a la metodología propuesta. En ambos casos se remarca que el refinamiento es correcto de acuerdo a la traducción propuesta. El protagonismo de éste concepto en la metodología implica que, la elección de una mala traducción, puede derivar en un escenario en el que un par de semánticas que “a ojo” parece configurar un buen refinamiento, no sea considerado como tal. Este problema es solucionable mientras se pueda buscar y definir una segunda traducción que exprese formalmente, el criterio utilizado para haber determinado “a ojo” la

correctitud del refinamiento. Se plantea entonces el siguiente interrogante: Dados dos comportamientos que "a ojo" no pueden ser uno un refinamiento del otro bajo ningún concepto, ¿Existirá una traducción que invalide este criterio y determine que, metodológicamente hablando, las semánticas configuran un refinamiento correcto? Es decir, ¿Una traducción apropiada puede "arreglar" un "refinamiento" que es decididamente erróneo? De ser afirmativa la respuesta la aplicación de la metodología podría resultar en falsos positivos, lo cual atentaría seriamente contra la aplicabilidad de la misma. Se intentará contestar esta pregunta en el transcurso de esta sección.

Para comenzar a responder el interrogante planteado, se propone analizar el siguiente ejemplo (cuyo código fuente se puede consultar en el listado `Turns_SharedVar.lts` del Apéndice B) en el que se presenta una pequeña descripción arquitectónica que hace uso del estilo *Shared Variable*:

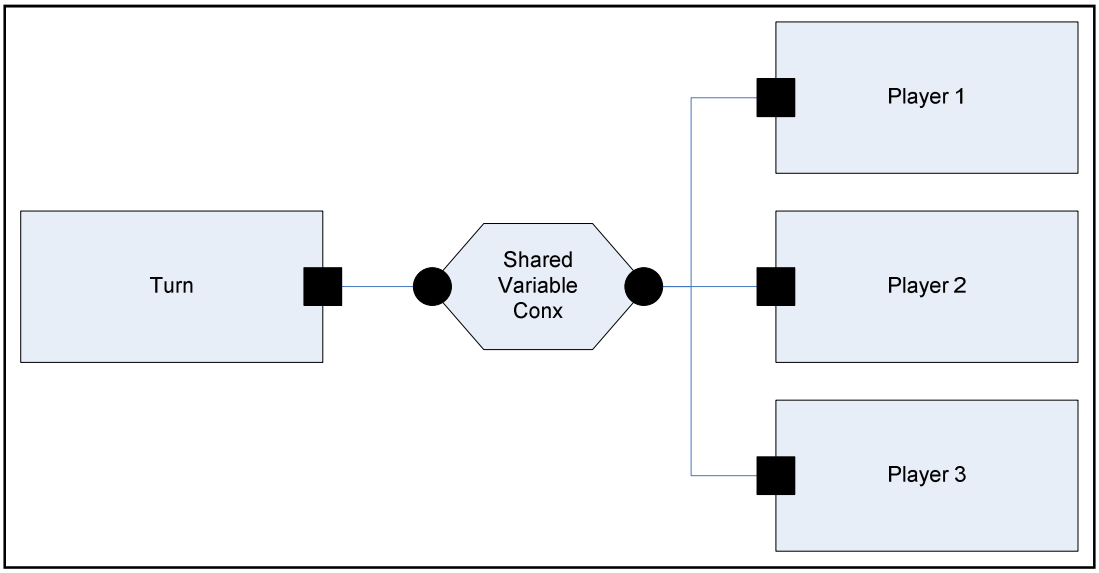

Figura 4-8: Descripción arquitectónica del reparto de turnos utilizando el estilo Shared Variable

La descripción modela un pequeño sistema en el que tres jugadores, numerados del uno al tres, se turnan para participar de un juego. Cada uno de los jugadores lee continuamente el valor de la *variable* "Turn" y si el valor leído es igual al número de jugador, éste determina que debe tomar su turno. Una vez que cada jugador da por concluido su turno, procede a escribir el número del próximo jugador en la *variable.* El estilo *Shared Variable* predica las siguientes propiedades:

```
property RWS_VARIABLE_BEHAVIOR = RWS_VARIABLE_BEHAVIOR[INITIAL_DATA_VALUE],
RWS_VARIABLE_BEHAVIOR[d:DATA] =
(
        rw[RWS].read[d] -> RWS_VARIABLE_BEHAVIOR[d]
        | rw[RWS].write[w:DATA] -> RWS_VARIABLE_BEHAVIOR[w]
).

property RW_BEHAVIOR(RW=1) =
(
        rw[RW].read[d:DATA] -> RW_BEHAVIOR | rw[RW].write[d:DATA] -> RW_BEHAVIOR
).
```

Se propone el comportamiento caracterizada por la función $\psi_{VariablePlayers}$ para la descripción presentada, con los siguientes procesos:

```
RW(DATA_MY_TURN=DATA_MIN, DATA_NEXT_TURN=DATA_MAX) =
(
        read[d:DATA] -> if (d == DATA_MY_TURN) then
                                        (turn -> write[DATA_NEXT_TURN] -> RW)
                                else
                                        RW
)
+ {read[DATA], write[DATA]}.

VARIABLE = VARIABLE[INITIAL_DATA_VALUE],
VARIABLE[d:DATA] =
(
        varRead[d] -> VARIABLE[d] | varWrite[dataWrite:DATA] -> VARIABLE[dataWrite]
).

VAR_CONX =
(
        rw[RWS].write[w:DATA] -> VAR_CONX
        | rw[RWS].read[d:DATA] -> VAR_CONX
).

|| ARQ =
(
        forall[rw:RWS] rw[rw]:RW(rw, (rw + 1) % (RWS_MAX + 1))
        || VAR_CONX
        || VARIABLE
                / { rw[RWS].write[w:DATA] / varWrite[w], rw[RWS].read[d:DATA] / varRead[d]}
).
```

- $\psi_{VariablePlayers}(Turn) =$ `VARIABLE`
- $\psi_{VariablePlayers}(Shared\ Variable\ Conx) =$`VAR_CONX`
- $\forall i: 1..3,\ \psi_{VariablePlayers}(Player\ i) =$ `rw[i]:RW(i, (i + 1) % (RWS_MAX + 1))`

Se puede verificar que el comportamiento satisface las propiedades del estilo utilizando la función identidad como traducción canónica.

El refinamiento propuesto para este caso (cuyo código fuente se puede consultar en el listado `Turns_ClientServer.lts` del Apéndice B), consiste en reemplazar el uso del estilo *Shared Variable* por el estilo *Client-Server.* De esta forma, el conector *Shared Variable Conx* se transforma en el conector *CS Conx* y la interacción entre los jugadores y la *Variable* cambia acorde al mismo. Se propone un nuevo comportamiento caracterizado por $\psi_{CSPlayers}$ con las siguientes definiciones:

```
// CS CONNECTOR
// ------------

CS_CONX_ST_STATUS(T=0) = CS_CONX_ST_STATUS_FREE,
CS_CONX_ST_STATUS_FREE =
(
        st[T].call[c:CLIENTS][CLIENT_CALLS] -> CS_CONX_ST_STATUS_BUSY[c]
),
CS_CONX_ST_STATUS_BUSY[c:CLIENTS] =
(
        st[SERVER_THREAD_BUSY].call[CLIENTS][CLIENT_CALLS] -> CS_CONX_ST_STATUS_BUSY[c]
```

```
        | st[T].response[c][SERVER_RESPONSES] -> CS_CONX_ST_STATUS_FREE
).

CS_CONX_CLIENT(C=0) =
(
        client[C].call[call:CLIENT_CALLS] ->
        (
                st[s:SERVER_THREADS].call[C][call] ->
                st[s].response[C][r:SERVER_RESPONSES] -> client[C].response[r]  ->
                CS_CONX_CLIENT
                |
                st[SERVER_THREAD_BUSY].call[C][call] -> CS_CONX_CLIENT
        )
).

|| CS_CONX =
(
        forall[t:SERVER_THREADS] CS_CONX_ST_STATUS(t)
        || forall[c:CLIENTS] CS_CONX_CLIENT(c)
)
/ {client[c:CLIENTS].dos / st[SERVER_THREAD_BUSY].call[c][CLIENT_CALLS]}.

// CS CLIENT (RW)
// --------------

CLIENT_RW(MY_TURN=DATA_MIN) = CLIENT_RW_POLL_TURN,
CLIENT_RW_POLL_TURN =
(
        call[CALL_READ] ->
        (
                dos -> CLIENT_RW
                |
                response[r:DATA_MIN..DATA_MAX] ->
                if (r == MY_TURN) then
                        (turn -> CLIENT_WRITE_MY_TURN)
                else
                        CLIENT_RW_POLL_TURN
        )
),
CLIENT_WRITE_MY_TURN =
(
        call[(MY_TURN + 1) % (RWS_MAX + 1)] ->
        (
                dos -> CLIENT_WRITE_MY_TURN
                | response[SERVER_RESPONSE_OK] -> CLIENT_RW
        )
)
+ {call[CLIENT_CALLS]}.

// CS SERVER (Variable Vault)
// --------------------------

SERVER_THREAD(T=SERVER_THREAD_MIN) =
(
        st[T].call[c:CLIENTS][CALL_READ] ->
        st[T].response[c][SERVER_RESPONSES_DATA] -> SERVER_THREAD
        |
        st[T].call[c:CLIENTS][w:CALL_WRITE] ->
        st[T].response[c][SERVER_RESPONSE_OK] -> SERVER_THREAD
)
+ {st[T].response[CLIENTS][SERVER_RESPONSES]}.

SERVER_VARIABLE = SERVER_VARIABLE[INITIAL_DATA_VALUE],
SERVER_VARIABLE[d:DATA] =
(
        st[SERVER_THREADS].call[CLIENTS][w:CALL_WRITE] -> SERVER_VARIABLE[w]
        | st[SERVER_THREADS].call[CLIENTS][CALL_READ] -> SERVER_VARIABLE[d]
        | st[SERVER_THREADS].response[CLIENTS][d] -> SERVER_VARIABLE[d]
        | st[SERVER_THREADS].response[CLIENTS][SERVER_RESPONSE_OK] -> SERVER_VARIABLE[d]
).
```

```
|| SERVER =
(
        forall[t:SERVER_THREADS] SERVER_THREAD(t)
        || SERVER_VARIABLE
).
```

- $\psi_{CSlayers}(Turn) =$ SERVER
- $\psi_{CSPlayers}(CS\ Conx) =$CS_CONX
- $\forall c: 1..3,\ \psi_{CSPlayers}(Player\ c) =$ `client[c]:CLIENT_RW / {rw[c].turn / client[c].turn}`

con ARQ $= BehaviorAsFSP(\psi_{CSPlayers})$:

```
|| ARQ =
(
        forall[c:CLIENTS] client[c]:CLIENT_RW(c) / {rw[c].turn / client[c].turn}
        || CS_CONX
        || SERVER
).
```

Para verificar que la nueva descripción satisface las propiedades interesantes de la original, se propone utilizar el proceso de traducción TRANSLATE_RW como mecanismo de traducción:

```
TRANSLATE_RW =
(
        st[SERVER_THREADS].call[c:CLIENTS][w:CALL_WRITE]
         -> rw[c].write[w] -> TRANSLATE_RW
        | st[SERVER_THREADS].response[c:CLIENTS][d:SERVER_RESPONSES_DATA]
         -> rw[c].read[d] -> TRANSLATE_RW
).
```

Mediante la composición de los procesos ARQ y TRANSLATE_RW se verifica que, con el mecanismo de traducción propuesto, el par de comportamientos determina un refinamiento válido. Hasta aquí no se encuentran problemas, por lo que sería conveniente crear alguno para mantener el interés del lector. Se crea un nuevo comportamiento determinado por $\psi_{CSPlayers2}$. La función $\psi_{CSPlayers2}$ es igual a a la función $\psi_{CSPlayers}$ exceptuando la definición del proceso *Server* que se redefine de la siguiente forma:

```
SERVER_THREAD(T=SERVER_THREAD_MIN) =
(
        st[T].call[c:CLIENTS][CALL_READ] ->
        st[T].response[c][SERVER_RESPONSES_DATA] ->
        SERVER_THREAD
        |
        st[T].call[c:CLIENTS][w:CALL_WRITE] ->
        st[T].response[c][SERVER_RESPONSE_OK] ->
        SERVER_THREAD
).

SERVER_VARIABLE = SERVER_VARIABLE[INITIAL_DATA_VALUE],
SERVER_VARIABLE[d:DATA] =
(
        st[SERVER_THREADS].call[CLIENTS][w:CALL_WRITE] -> SERVER_VARIABLE[w]
```

```
        | st[SERVER_THREADS].call[CLIENTS][CALL_READ] -> SERVER_VARIABLE[d]
        | st[SERVER_THREADS].response[CLIENTS][DATA_MIN] -> SERVER_VARIABLE[d]
        | st[SERVER_THREADS].response[CLIENTS][SERVER_RESPONSE_OK] -> SERVER_VARIABLE[d]
)
+ {st[SERVER_THREADS].response[CLIENTS][SERVER_RESPONSES]}.

|| SERVER =
(
        forall[t:SERVER_THREADS] SERVER_THREAD(t)
        || SERVER_VARIABLE
).
```

Es decir, se modifica el componente *Turn* para que toda respuesta a una llamada de lectura por parte de los jugadores (`CALL_READ`) sea respondida siempre con `DATA_MIN`. Claramente, este cambio debería "romper" el ejemplo. Manteniendo el mecanismo de traducción propuesto, no se verifica que el comportamiento determinado por $\psi_{CSPlayers2}$ sea un refinamiento válido del comportamiento determinado por $\psi_{VariablePlayers}$ (ya que ahora la composición de los procesos `ARQ` y `TRANSLATE_RW` no satisface la propiedad `RWS_VARIABLE_BEHAVIOR` definida por el estilo *Shared Variable*). ¿Sera posible proveer un mecanismo de traducción por el cual se determine lo contrario?

Reescribiendo el proceso de traducción de la siguiente forma:

```
TRANSLATE_RW2 = TRANSLATE_RW[INITIAL_DATA_VALUE],
TRANSLATE_RW2[d:DATA] =
(
        st[SERVER_THREADS].call[c:CLIENTS][w:CALL_WRITE] ->
        rw[c].write[w] ->
        TRANSLATE_RW2[w]
        |
        st[SERVER_THREADS].response[c:CLIENTS][response:SERVER_RESPONSES_DATA] ->
        if (response == d) then (rw[c].read[d] -> TRANSLATE_RW[d]) else TRANSLATE_RW2[d]
).
```

y componiéndolo adecuadamente, resulta en la verificación del refinamiento (LTSA no reporta la violación de ninguna propiedad del estilo refinado). Se "arregló" un mal refinamiento mediante una traducción "malvada". ¿Se debe dejar de leer inmediatamente este trabajo ya que se demostró la existencia de un falso positivo? El autor le propone al lector que no lo haga; existen dos problemas con el contra-ejemplo propuesto y se analizará cada uno de ellos a continuación.

El primer problema es el resultado de una suerte de irresponsabilidad en el uso de la traducción propuesta. Si se analiza el proceso de traducción, se podrá notar que éste "expresa" claramente que solo se considera como una *lectura* de la *variable* aquellas respuestas cuyo valor sea igual al último valor escrito en la *variable*. El uso de un mecanismo de traducción implica que se está de acuerdo con lo que el mismo expresa. En este caso se está declarando que en la descripción arquitectónica que hace uso del estilo *Client-Server,* solo se consideran como *lecturas* un subconjunto (conveniente) de las respuestas provistas por el componente que hace las veces de *Server*. En resumen, quien propone un mecanismo de traducción debe hacerse cargo de las implicaciones de su propuesta. Más allá de éste potencial mea culpa, el problema queda pendiente: la metodología indica que el refinamiento es válido. Si se analiza con atención, el

"positivo" que resulta de la aplicación de la metodología se traduce como: "La descripción arquitectónica más concreta, efectivamente se comporta como dicta el estilo *Shared Variable",* lo cual es cierto. La descripción arquitectónica más concreta se comporta como una instancia "viciada" del estilo *Shared Variable* en el cual solo se realizan *lectura*s de un único valor. Quien utilice ésta traducción deberá comprender que está proponiendo que no todas las *respuestas* del servidor ante un *llamado* `CALL_READ` sean equivalentes a una *lectura* de la *variable*. La semántica $\psi_{CSPlayers2}$ parecería ignorar esta propuesta e interpreta todas las *respuestas* como *lecturas*. Sin embargo, como no se propuso ninguna propiedad que predique sobre el uso que se le da a la *variable* leída*,* la metodología no puede determinar que no se está honrando la traducción. De esta forma se introduce el segundo problema: la sub-especificación.

El segundo problema es que la descripción original se encuentra sub-especificada. En ningún momento se definió ninguna propiedad de sistema (no de estilos) para la misma, ni siquiera se definió una propiedad que garantice el orden de los jugadores, como la exhibida a continuación:

```
property TURNS = TURNS[RWS_MIN],
TURNS[t:RWS] = (rw[t].turn -> TURNS[(t + 1) % (RWS_MAX + 1)]).
```

Las propiedades de sistema cumplen un rol sumamente importante en la metodología: al especificar cómo se instancian los estilos utilizados, completan el comportamiento de las descripciones reduciendo la posibilidad de toparse con traducciones "malvadas". Si las propiedades de sistema incluyen acciones que no pertenecen a ninguno de los lenguajes utilizados para implementar un estilo (es decir, pertenece al lenguaje propio del sistema), la protección será todavía mayor, ya que estas acciones no pueden ser manipuladas mediante un mecanismo de traducción. Se puede verificar que el comportamiento determinada por $\psi_{CSPlayers}$ satisface la propiedad `TURNS` utilizando el proceso de traducción `TRANSLATE_RW`, mientras que $\psi_{CSPlayers2}$ con el proceso de traducción `TRANSLATE_RW2` no lo hace. La propiedad `TURNS` provee información suficiente sobre el comportamiento pretendido del sistema para determinar que el par [proceso de traducción `TRANSLATE_RW2`; comportamiento determinada por $\psi_{CSPlayers2}$], no configura un buen refinamiento para la descripción original (porque el comportamiento no hace honor a la propuesta de la traducción).

Del ejemplo presentado en esta sección se pueden extraer las siguientes moralejas:

1. La calidad de la especificación del comportamiento esperado para un sistema determina una cota máxima en la calidad del refinamiento: La metodología valida refinamientos **con respecto a un grupo de propiedades que debe proveerse explícitamente**. Si una propiedad no es explicitada, entonces la metodología no puede garantizar su cumplimiento en la descripción más concreta.

2. El uso de un mecanismo de traducción implica un compromiso y los comportamientos propuestos deben honrarlo. Los mecanismos de traducción pueden ser procesos

complejos con estado propio y los comportamientos con los que se pretenda utilizarlos deben reflejar esta complejidad.

3. Las propiedades de sistemas son sumamente importantes ya que reifican la forma en que los estilos son utilizados y, de esta forma, aportan información vital sobre el comportamiento esperado de un sistema.

4.4 Uso de LTSA

Uno de los objetivos de la metodología propuesta es proveer la posibilidad de reutilizar conocimiento mediante el uso de estilos arquitectónicos, para lograr la manifestación del fenómeno de económica cognitiva. LTSA y FSP en conjunto, tal como se presentan en [MK99], no proveen mecanismos adecuados para la reutilización de código. De hecho, se puede afirmar que el único mecanismo provisto es la parametrización de definiciones de procesos y propiedades. En consecuencia, es necesario apelar a mecanismos alternativos para lograr el objetivo propuesto de reutilizar conocimiento (codificado, en este caso, mediante FSP). Estos mecanismos alternativos fueron aplicados para definir los estilos arquitectónicos de tal forma, que las definiciones puedan ser reutilizadas en el marco de la instanciación LTSA de la metodología (Es decir, la misma definición del estilo *Client-Server* debería poder ser utilizada para todas las descripciones arquitectónicas y comportamientos asociados).

El primer mecanismo alternativo es el uso de rangos y constantes en la definición de los estilos. Alterando el valor definido para cada uno de éstos, se afecta implícitamente la definición de las propiedades que los utilizan. De esta forma, se logra un mecanismo de parametrización que afecta a todas las definiciones propuestas por un estilo (Es decir, se parametriza el estilo). Las definiciones de estilos provistas en este trabajo resultan entonces, en un conjunto de propiedades (posiblemente parametrizables) que utilizan los valores de los rangos y constantes establecidos por el estilo para describir el comportamiento de los componentes y conectores (parametrizandolos implícitamente). Por ejemplo, en la definición del estilo *Client-Server* se utiliza el rango `CLIENTS` para numerar y diferenciar a cada uno de los *Clientes*.

La definición de un estilo predica propiedades sobre una única instanciación del mismo: continuando con el ejemplo del estilo *Client-Server*, las propiedades definen el comportamiento de un único *Servidor* con un único *Conector* y un número de *Clientes* definidos por el rango `CLIENTS`. Dado que una descripción puede exhibir más de una instanciación del mismo estilo, se apela a los operadores de prefijo y renombre: Por ejemplo, en una descripción que exhibe el estilo *Pipe+Filter* habrá, generalmente, múltiples *Pipes* y múltiples *Filtros*. Cada uno de ellos es una instanciación del estilo que debe ser verificada por las propiedades definidas. Se utiliza los operadores de prefijo y renombre para obtener múltiples “copias” de las propiedades definidas por el estilo. Cada copia será aplicada a una única instancia del estilo en la descripción. El ejemplo de Monte Carlo, presentado anteriormente en este capítulo, hace uso de este recurso. Para garantizar que los cambios introducidos en el lenguaje mediante los operadores de prefijo y renombre no modifiquen la semántica original del estilo, basta con encontrar una función biyectiva que relacione el alfabeto original con el nuevo lenguaje obtenido. La aplicación de este

recurso de reutilización de código, no debe confundirse con el concepto de traducción canónica que es una forma de reificar la adaptación de un lenguaje a otro.

Los valores establecidos para los rangos y constantes afectan a todas las "copias" creadas mediante los operadores de renombre y prefijo (ya que en LTSA las constantes y los rangos tienen visibilidad global). Si se necesitan dos "copias" de la especificación del estilo, para dos instanciaciones que difieren en alguno de los aspectos definidos por los rangos o constantes (por ejemplo dos instanciaciones de *Client-Server,* cada una con un número de *clientes* diferente), se puede crear (y luego aplicar) una nueva definición de la siguiente forma:

1. Se copia (a mano) la definición original
2. Se reemplazan las definiciones de constantes y rangos por nuevas definiciones con nuevos nombres
3. Se reemplazan las referencias a las constantes y rangos originales por los definidos en el punto anterior
4. Se cambian los nombre de las propiedades para no provocar colisiones con las propiedades de la definición original

Dado que los mecanismos de reutilización de código utilizados son engorrosos y propensos a errores, se propone como trabajo futuro la creación un conjunto de herramientas que permita, entre otras cosas, automatizar estas tareas al ocultar o simplificar el uso de LTSA.

4.5 Limitaciones del enfoque actual

Mediante [Brooks87] Brooks enfrenta a la comunidad de desarrollo de software con la más horrible de las verdades: No existe una bala de plata (O en versión criolla, a falta de hombres lobos en nuestro folklore, no existen soluciones mágicas). Una de las soluciones (no mágicas) propuestas por Brooks para combatir la triste realidad, es contar con "Great Designers". Bach (que es J de James y no de Johann Sebastian) suplica a la comunidad para que se aliente la aparición de "Project Heroes" en [Bach95]. En ambos casos se intenta establecer que la educación / capacitación / experiencia (en la materia, claro está) de la gente que desarrolla software es la mejor protección contra el fracaso: El conocimiento es poder.

Una interpretación libre de [Brooks87] permite llegar a la siguiente máxima: Es necesario sincerarse sobre las limitaciones que cada herramienta / técnica / metodología lleva aparejada (ya que ninguna es la bala de plata). Conocer las limitaciones de las técnicas y herramientas con las que se cuenta para el desarrollo de software, permite seleccionar, aplicar y modificar cada una de ellas en el momento oportuno, maximizando los beneficios que resultarán de su uso. Ateniéndose a la máxima citada, se analizan las limitaciones de la instanciación de la metodología propuesta. Para cada unas de ellas se propone una potencial solución en el capítulo "Trabajo Futuro" de este trabajo.

La primera limitación es que la metodología se enfoca exclusivamente sobre los aspectos de comportamiento de las descripciones: No existe la posibilidad de enunciar propiedades estructurales como "El *Cliente* y el *Servidor* deben ser componentes separados" ya que la

metodología no propone ninguna herramienta que relacione las estructuras (componentes, conectores y su configuración) potencialmente diferentes de dos descripciones.

Otra limitación, es la restricción en el tamaño de los modelos. A modo de ejemplo, se presenta la siguiente tabla donde figura la cantidad de estados y transiciones para las descripciones concreta y abstracta del ejemplo de la agencia de noticias, en función de la cantidad de redacciones de noticias:

Redacciones de Noticias	Estados		Transiciones	
	Announcer-Listener	Client-Server	Announcer-Listener	Client-Server
1	2.208	30.375	11.608	198.153
2	7.248	211.869	36.312	1.127.355
3	19.704	1.231.089	94.024	5.186.405
4	101.952	N/A	647.168	N/A
5	522.576	N/A	4.141.536	N/A

Con cuatro o más redacciones de noticias, la versión de 32 bits de LTSA no puede componer la descripción que utiliza el estilo *Client-Server* por falta de memoria. Se puede apreciar que la cantidad de estados y transiciones crecen exponencialmente, limitando severamente el tamaño de los modelos en la práctica. Estas limitaciones son un resultado directo de la utilización de herramientas de model checking [CES86].Como resultado se debe trabajar, generalmente, sobre simplificaciones de la arquitectura real, utilizando sets de datos limitados o reduciendo el número de entidades del modelo (*Clientes* en una descripción *Client-Server*, *Eventos* y *Listeners* en una descripción *Announcer-Listener*, etc.). En los capítulos 8 y 11 de [MK99] se presentan diferentes propuestas parar sobrellevar las limitaciones de LTSA (propuestas que fueron aplicadas a lo largo de este trabajo). Este problema se agrava debido a la definición de mecanismo de traducción utilizada en la instanciación con LTSA. Dado que ésta se basa en la composición de procesos, implica necesariamente un aumento exponencial del número de estados y transiciones del modelo.

La última limitación (aunque seguramente el tiempo y los críticos bien o mal intencionados encontrarán muchas más) es que la propuesta se restringe únicamente al trabajo con arquitecturas de naturaleza estática. Para una arquitectura *Client-Server*, por ejemplo, el número de *clientes* debería poder variar en el tiempo; un *servidor* web no atiende un número constante de *clientes* todo el tiempo. Existe una solución simple para estos casos: se define un número máximo X de *clientes*, si se desea trabajar con menos *clientes* se programa el modelo para que algunos de éstos "se pongan a dormir" (no interactúen con el servidor u otros procesos) regularmente. De esta forma se "simula" el dinamismo de una arquitectura, pero la metodología no provee ningún

"soporte nativo" para estos casos (como podría ser, formalismos para modelar cambios de configuración).

5. Trabajo Futuro

El objetivo de esta tesis es presentar una metodología que permita determinar formalmente si una descripción arquitectónica es un refinamiento correcto de otra. Detrás de éste se esconde un segundo objetivo mucho más ambicioso: presentar una visión de desarrollo de software en la cual las descripciones arquitectónicas juegan un rol central para la comunicación, el análisis y la construcción de sistemas complejos. De concretarse esta visión, el concepto de refinamiento ocuparía un rol central como elemento de trazabilidad entre descripciones que exhiben diferentes niveles de abstracción.

La primera propuesta de trabajo futuro, es la creación de herramientas que permitan aplicar la metodología sin necesidad de contar con el bagaje teórico incluido en este trabajo. Por ejemplo, para aplicar la metodología de refinamientos con LTSA se debería contar con:

- Un ADL apto para describir vistas conceptuales, que permita asociar comportamiento a los elementos descriptos. Dicho comportamiento podría especificarse como procesos FSP o mediante un lenguaje que luego es compilado a FSP. ACME [GMW00] es un ejemplo de un ADL que cumple con estos requisitos

- Un editor para el ADL elegido que, además, permita definir y editar el comportamiento asociado a los elementos, las propiedades a verificar y los mecanismos de traducción. Junto con ACME se puede utilizar ACMEStudio [SG04], con funcionalidad extendida mediante plug-ins específicos

- Si se utiliza FSP como lenguaje de descripción de comportamiento, una herramienta que valide que los procesos, propiedades y mecanismos de traducción se adecuen a las restricciones impuestas por la metodología. Si se utiliza otro lenguaje, un compilador que produzca código LTSA adecuado para los procesos, propiedades y traducciones a partir de éste

- Una herramienta que dadas: dos descripciones arquitectónicas con su comportamiento, un conjunto de propiedades a validar y un mecanismo de traducción, utilice LTSA para componer los procesos FSP adecuadamente y así determinar si el par de descripciones configuran un refinamiento válido, de acuerdo a las propiedades y traducciones propuestas.

Mediante la creación de herramientas apropiadas, se debería facilitar la adopción y aplicación de la metodología y del concepto de refinamiento.

Otra línea de trabajo plausible es investigar el uso de diferentes formalismos para la especificación del comportamiento de las arquitecturas, con dos objetivos: El primero es buscar formalismos más accesibles que LTSA para el público general. El segundo es intentar superar las limitaciones que impone el model checking tradicional en cuanto al tamaño y complejidad de los modelos. Una de

las opciones más interesantes a este respecto es el uso de formalismos que permiten crear descripciones de comportamiento parciales, como por ejemplo MTSA [UBC07]. Las descripciones arquitectónicas son parciales por definición, por lo que el uso de estos formalismos permitiría describirlas más "naturalmente". Como resultado, se debería reconsiderar la clasificación de las propiedades de una descripción en deseables y accidentales, tal como se presento en el trabajo. Idealmente, se podría trabajar bajo la premisa de que solo las propiedades deseables son las descriptas.

El estudio de diferentes mecanismos de traducción y de las restricciones que deben aplicarse a los mismos, también es un área que puede ser profundizada. En este trabajo se restringe los mecanismos de traducción a partir de las nociones de lenguaje de sistema y lenguaje de estilos. Los mecanismos no pueden predicar sobre el lenguaje de sistema de la descripción más abstracta. De esta forma, no se permiten refinar conceptos propios (no de los estilos utilizados) de dicha descripción. Proponer mecanismos que permitan refinar estos conceptos permitiría ampliar las posibilidades de refinamiento, al permitir expresar ideas propias de la teoría del sistema a diferentes niveles de abstracción.

También dentro del estudio de los mecanismos de traducción, se debería determinar si es posible utilizar la información provista por el mecanismo de traducción propuesto para facilitar la tarea de comprobar la validez de las propiedades del sistema en la traducción. La intuición es que el mecanismo de traducción provee cierta información contextual que, junto a la solución propuesta en la descripción más abstracta, permiten establecer por adelantado ciertas propiedades de la traducción. Estas propiedades deberían facilitar la tarea de comprobar las propiedades de sistema en la traducción, al hacer las veces de pre-condiciones.

La aplicabilidad de la metodología debe ser validada mediante "trabajo de campo" que permita estudiar el grado de adopción que la misma logra, junto a las ventajas y desventajas que supone su uso en ámbitos no estrictamente académicos. A partir del "trabajo de campo", se podría crear un catálogo de refinamientos arquitectónicos que capture los rationales de refinamiento que se repiten en la comunidad y que conforman el conocimiento común en la materia. Además se podría validar la generalidad de las formalizaciones propuestas para los estilos (detalladas en el apéndice "Catálogo de Estilos"). Una de las lecciones más valiosas que dejó [GHJV95], es que el éxito de cualquier tipo de catálogo para el desarrollo de software depende de su capacidad para reflejar y generalizar la realidad. Un catalogo no debe ser un intento de imponer una realidad solo existente en la mente de un autor.

El "trabajo de campo" permitiría conocer en qué etapas del proceso de desarrollo y con qué fines se apela al uso de la metodología propuesta. Se debería poder determinar, entonces, como "encaja" la propuesta de este trabajo en las diferentes metodologías de desarrollo practicadas por la comunidad. Luego se podría comprobar si los beneficios vaticinados efectivamente se manifiestan en la práctica. De esta forma sería posible contestar preguntas como ¿Las jerarquías de descripciones arquitectónicas se utilizan realmente para transmitir la teoría del sistema en la práctica más allá de su aparente aptitud para ello?

Finalmente, sería interesante estudiar cómo puede combinarse la metodología propuesta con otras concepciones de refinamiento arquitectónico. Existen varios trabajos que estudian los refinamientos desde puntos de vista puramente estructurales e incluyen la posibilidad de trabajar con arquitecturas dinámicas [AM99, CPT99]. Estos podrían utilizarse junto a la metodología propuesta para estudiar en forma conjunta todos los aspectos del concepto de refinamiento (Aspectos de comportamiento, estructurales, composicionales, etc.).

6. Conclusiones

El uso de descripciones arquitectónicas es uno de los conceptos del estudio de las arquitecturas de software que mayor adopción ha logrado en el desarrollo práctico de sistemas. Las descripciones arquitectónicas se utilizan para transmitir conceptos propios de un sistema mediante herramientas que exhiben un nivel de abstracción adecuado para tal fin. La complejidad natural de todo sistema [Brooks87], resulta en la necesidad de contar con más de una descripción para transmitir los conceptos asociados al mismo. Si dos descripciones arquitectónicas describen el mismo sistema utilizando diferentes niveles de abstracción, entonces se debería poder encontrar cierta trazabilidad entre éstas. Esta es la función del concepto de refinamiento arquitectónico: reificar la trazabilidad entre descripciones con diferentes niveles de abstracción.

Una definición adecuada de refinamiento permite crear, a partir de descripciones existentes, nuevas descripciones que exhiben cambios sofisticados en el nivel de abstracción y que determinan cambios igualmente sofisticados en la información exhibida. El resultado es la obtención de una "nueva forma de interpretación" del sistema modelado. De esta manera, se puede crear una jerarquía de descripciones arquitectónicas que describen al sistema mediante conceptos heterogéneos. La trazabilidad existente en la jerarquía, permite que los conceptos exhibidos por cada una de las descripciones arquitectónicas se configure en una teoría unificada del sistema. Como resultado, se logra obtener un "mapa" conceptual del sistema por el cual es posible determinar el origen de cada una de las abstracciones utilizadas. Este mapa es una representación fiel de los procesos mentales realizados por los desarrolladores para "pensar el sistema". Así, el concepto de refinamiento permite exhibir la teoría del sistema en una forma eficiente y práctica. Contar con una definición más poderosa de refinamiento, permite obtener jerarquías más complejas y por lo tanto, maximizar los beneficios mencionados en este párrafo.

En este trabajo se presentó una definición de refinamiento arquitectónico que permite relacionar descripciones de la vista conceptual que no configuran un refinamiento estrictamente composicional y jerárquico (el tipo de refinamiento más común hasta el momento). Esta definición se obtuvo mediante un estudio previo del concepto que permitió considerar adecuadamente el contexto de su aplicación: el marco de trabajo arquitectónico. A partir de éste, se desarrolló una metodología aplicable en la práctica mediante LTSA. Los ejemplos presentados confirman que la definición de refinamiento presentada, permite considerar refinamientos complejos que implican la relación de conceptos heterogéneos. Un segundo punto demostrado mediante los ejemplos, es que la expresividad provista por FSP es suficiente para modelar los estilos arquitectónicos más comunes y sus mecanismos. Entonces, es plausible creer que una metodología como la propuesta es aplicable en la práctica sobre modelos simplificados de sistemas y que reportará todos los beneficios asociados al concepto de refinamiento. Estos beneficios se verán maximizados gracias a la definición presentada, que permite trabajar con refinamientos complejos

Muchos trabajos sobre arquitecturas de software comienzan con un ejemplo en el cual se comparan dos potenciales descripciones arquitectónicas para un mismo sistema. Dado que cada

descripción hace uso de estilos diferentes, se analizan las ventajas y desventajas que resultarán de la elección de cualquiera de las dos descripciones. A partir del resultado del análisis, se elige una descripción y se descarta la otra. Contando con el concepto de refinamiento tal como se definió en este trabajo, no es necesario descartar ninguna de las dos descripciones. Mientras ambas descripciones sean útiles y trazables a un mismo concepto (es decir pertenezcan a una misma jerarquía de refinamiento), las dos servirán como descripciones del el sistema. Cada una exhibirá un conjunto diferente de propiedades que resultará de los estilos aplicados, cada una será de utilidad en algún aspecto del desarrollo. De esta forma, las descripciones se complementan en vez de "competir" entre ellas. La aplicación del concepto de refinamiento habilita entonces, la posibilidad de describir sistemas complejos con mucha más facilidad gracias a la capacidad de complementar ideas complejas. Mientras más poderosa sea la definición de refinamiento con la que se trabaja, más complejos serán los sistemas que se puedan describir.

Los aportes realizados por este trabajo son:

1. La creación de un marco teórico-formal que justifica la existencia del concepto

2. Una definición de refinamiento que resulta aplicable en la práctica y que permite trabajar con refinamientos complejos

3. Un uso práctico del concepto de estilos arquitectónicos, en el cual se reconoce explícitamente la diferencia entre un estilo y la implementación del mismo

Finalmente, como ocurre con todo concepto o herramienta que se pretende sea útil en la práctica, solo la experiencia empírica determinará la idoneidad de la propuesta y confirmara o desmentirá las conclusiones presentadas.

7. Apéndice A – Catálogo de Estilos

En este apéndice, se presentan las formalizaciones mediante FSP de los estilos arquitectónicos utilizados en el transcurso de este trabajo. Se puede considerar este apéndice como una reedición de [AAG95] utilizando LTSA y los conceptos arquitectónicos consolidados en los 13 años que separan este trabajo de aquel. A diferencia del trabajo citado, las formalizaciones aquí presentadas se concentran en los aspectos de comportamiento y no en los estructurales (de acuerdo a las necesidades de este trabajo). Es necesario aclarar que, de ninguna manera, se consideran las formalizaciones aquí presentadas como definiciones universales que deben ser adoptadas a rajatabla: Las especificaciones presentadas responden a las necesidades de modelado propias del trabajo y de las herramientas aplicadas. Es esperable que en la práctica existan múltiples formalizaciones, cada una adecuada para un marco de trabajo determinado. Las formalizaciones aquí presentadas, son las adecuadas para este trabajo en particular.

La presentación de cada uno de los estilos se estructurará de la siguiente forma:

- **Nombre:** El nombre del estilo especificado
- **Descripción informal:** Una descripción informal del estilo, que remite al "folklore" que rodea al mismo
- **Componentes:** Presentación y detalle de cada uno de los tipos de componentes propuestos por el estilo
- **Conectores:** Presentación y detalle de cada uno de los tipos de conectores propuestos por el estilo
- **Configuraciones legales:** Definición de las reglas que deben respetarse para interconectar los componentes y conectores presentados en los puntos anteriores
- **Formalización:**
    - **Estrategia de formalización:** Descripción de los conceptos aplicados para formalizar el estilo (de ser necesaria para la comprensión de las propiedades)
    - **Propiedades:** Las propiedades definidas por el estilo. Estas propiedades serán especificadas de acuerdo a lo presentado en el capítulo 4 de este trabajo. Junto a las propiedades se definirán también los fluents utilizados por las mismas
    - **Configuración formal:** Definición de la configuración formal de acuerdo a lo presentado en el capítulo 4 de este trabajo

- **Sub-Estilos:** Presentación de los diferentes sub-estilos posibles, de acuerdo a la adopción o no de las propiedades enumeradas

7.1 Estilo Announcer-Listener

Nombre: Announcer-Listener (también conocido como Publisher-Subscriber)

Descripción informal:

El estilo Announcer-Listener determina un esquema de notificación de eventos. El componente announcer emite eventos que los listeners recibirán si se encuentran suscriptos al momento del anuncio. Cada listener puede optar por suscribirse o desuscribirse a la recepción de eventos en cualquier momento y en forma independiente de los demás.

Componentes:

- **Announcer:** Los announcers son componentes cuyo único comportamiento consiste en la emisión periódica de eventos

- **Listener:** Los listeners son componentes cuyo comportamiento consiste en recibir los eventos emitidos por un announcer determinado. Los listeners explicitan su deseo de recibir eventos o no, suscribiéndose o desuscribiendose a los mismos

Conectores:

- **Conector Announcer-Listener:** Es el único conector existente en este estilo. Consta de dos roles, el de listeners y el del announcer a los que se conectan los componentes homónimos. El conector abstrae los aspectos de comunicación entre las partes al determinar que eventos deben ser recibidos por cada uno de los listeners de acuerdo al estado de su suscripción.

Configuraciones legales:

Las configuraciones legales quedan determinadas por los roles definidos por el único conector del estilo (un único announcer – múltiples listeners asociados a cada conector).

Formalización:

- **Estrategia de formalización:** La formalización propuesta para el estilo se basa en la tarea de abstracción del conector del estilo. El conector mantiene una cola de eventos pendientes para cada uno de los listeners. Cuando un listener recibe un evento, el

conector lo elimina de la cola. Cada vez que el announcer emite un evento, el conector se encarga de encolar dicho evento en cada una de las colas de los listeners suscriptos. De esta forma, el announcer no debe preocuparse por la cantidad o el estado de los listeners

- **Propiedades:** El estilo define tres propiedades que predican sobre los comportamientos esperados para los listeners (mediante la propiedad `LISTENER_BEHAVIOR`), los announcers (mediante la propiedad `ANNOUNCER_BEHAVIOR)` y sobre los eventos que deben ser recibidos por cada listener (mediante la propiedad `CORRECT_EVENTS`). Nótese que la propiedad `ANNOUNCER_BEHAVIOR` es trivialmente cierta y solo se incluye para exhibir lo trivial del comportamiento de los announcers. Se utilizan los rangos `LISTENERS` para definir al conjunto de listeners y `DATA` para definir los eventos que pueden ser anunciados

```
fluent FLUENT_LISTENER_SUBSCRIBED[l:LISTENERS] =
<{listener[l].subscribe},{listener[l].unsubscribe}>
```

```
property ANNOUNCER_BEHAVIOR = (announce[DATA] -> ANNOUNCER_BEHAVIOR).
```

```
property LISTENER_BEHAVIOR(L=LMIN) =
(
    listener[L].subscribe -> LISTENER_BEHAVIOR_SUBSCRIBED
),
LISTENER_BEHAVIOR_SUBSCRIBED =
(
    listener[L].event[DATA] -> LISTENER_BEHAVIOR_SUBSCRIBED
    | listener[L].unsubscribe -> LISTENER_BEHAVIOR
).
```

```
property CORRECT_EVENTS(L=LMIN) =
(
    listener[L].subscribe -> CORRECT_EVENTS_SUBSCRIBED
    | announce[DATA] -> CORRECT_EVENTS
),
CORRECT_EVENTS_SUBSCRIBED =
(
    listener[L].unsubscribe -> CORRECT_EVENTS
    | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[d]
),
CORRECT_EVENTS_SUBSCRIBED[e1:DATA] =
(
    listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED
    | listener[L].unsubscribe -> CORRECT_EVENTS
    | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[e1][d]
),
CORRECT_EVENTS_SUBSCRIBED[e1:DATA][e2:DATA] =
(
    listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED[e2]
    | listener[L].unsubscribe -> CORRECT_EVENTS
    | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[e1][e2][d]
),
CORRECT_EVENTS_SUBSCRIBED[e1:DATA][e2:DATA][e3:DATA] =
(
    listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED[e2][e3]
    | listener[L].unsubscribe -> CORRECT_EVENTS
).
```

- **Configuración formal:** La configuración formal está dada por $\{listener_l; announcer\}$, con $listener_l$ = {`listener[l].*`} para `l:LISTENERS` y $announcer$ = {`announce[DATA]`}

- **Sub-estilos:** Las propiedades definen un sub-estilo particular en el que los eventos no pueden perderse y por lo tanto si un listener no está dispuesto a consumir los eventos anunciados, eventualmente las colas de los conectores alcanzaran su tamaño máximo y el announcer no podrá realizar nuevos anuncios. Diferentes sub-estilos se pueden especificar al reemplazar la propiedad `CORRECT_EVENTS` por otras que prediquen diferentes reglas de envio y buffering de eventos

7.2 Estilo Client-Server

Nombre: Client-Server

Descripción informal:

El estilo Client-Server define una relación asimétrica entre componentes por la cual un conjunto de clientes independientes entre sí realiza llamadas a un servidor. El servidor atiende las llamadas en forma individual y concurrente. La comunicación siempre es iniciada por un cliente

Componentes:

- **Cliente:** Los clientes son componentes independientes entre sí que realizan llamadas al servidor en forma periódica y quedan a la espera de una respuesta del mismo como resultado

- **Servidor:** El servidor recibe las llamadas de los clientes y las procesa individualmente (y posiblemente, concurrentemente también) para retornar una respuesta. El servidor espera pasivamente por las llamadas de los clientes, no inicia comunicaciones hacia los clientes.

Conectores:

- **Conector Cliente-Servidor:** Es el único conector existente en este estilo. Consta de dos roles, el de clientes y el del servidor a los que se conectan los componentes homónimos. El conector abstrae los aspectos de comunicación entre los clientes y el servidor.

Configuraciones legales:

Las configuraciones legales quedan determinadas por los roles definidos por el único conector del estilo (un único servidor – múltiples clientes asociados a cada conector).

Formalización:

- **Estrategia de formalización:** La formalización propuesta para el estilo se basa en la existencia de una serie de hilos de ejecución (threads) independientes en el servidor. Cuando un cliente realiza un llamado, el conector establece una asociación temporal entre el cliente y uno de los threads del servidor. Una vez que el thread determina una respuesta, el conector la hace llegar al cliente y la asociación cliente-thread se elimina. Dado que pueden existir más clientes que threads, el conector debe tener la capacidad de encolar pedidos a la espera de la asignación de un thread. Si la cola alcanza un tamaño máximo, entonces el conector deberá rechazar los llamados de los clientes mediante un mensaje de "denial of service".

- **Propiedades:** El estilo define ocho propiedades que predican sobre los comportamientos esperados para los clientes (mediante la propiedad `CLIENT_BEHAVIOR`), los threads de los servidores (mediante la propiedad `SERVER_THREAD_BEHAVIOR)` y las asociaciones que se realicen entre clientes y threads (mediante las propiedades restantes). Se utilizan los rangos `CLIENTS` y `SERVER_THREADS` para definir los conjuntos de clientes y threads respectivamente y los rangos `CLIENT_CALLS` y `SERVER_RESPONSES` para definir las llamadas y respuestas

```
fluent SERVER_THREAD_BUSY[t:SERVER_THREADS] =
       <{associate[t][CLIENTS]}, {disassociate[t][CLIENTS]}>
```

```
property CLIENT_BEHAVIOR = (call[CLIENT_CALLS] -> (response[SERVER_RESPONSES] ->
       CLIENT_BEHAVIOR | dos -> CLIENT_BEHAVIOR)).
```

```
property SERVER_THREAD_BEHAVIOR(T=0) = (st[T].clientCall[CLIENT_CALLS] ->
st[T].clientResponse[SERVER_RESPONSES] -> SERVER_THREAD_BEHAVIOR).
```

```
property SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR(T=0) = (associate[T][c:CLIENTS] ->
st[T].clientCall[CLIENT_CALLS] -> st[T].clientResponse[SERVER_RESPONSES] ->
disassociate[T][c] -> SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR).
```

```
assert ASSOCIATE_NOT_BUSY_THREADS = forall[t:SERVER_THREADS]
([](X(associate[t][CLIENTS]) -> !SERVER_THREAD_BUSY[t]))
```

```
assert DOS_ONLY_IF_ALL_TRHEADS_ARE_BUSY = forall[c:CLIENTS] ([](X(client[c].dos) ->
forall[t:SERVER_THREADS] SERVER_THREAD_BUSY[t]))
```

```
ASSOCIATION_KEEPS_CLIENT_CALL(C=0) =
(
    client[C].call[d:CLIENT_CALLS] -> ASSOCIATION_KEEPS_CLIENT_CALL_1[d]
    | client[C].dos -> ERROR
```

```
     | associate[s:SERVER_THREADS][C] -> ERROR
     | st[SERVER_THREADS].clientCall[CLIENT_CALLS] -> ASSOCIATION_KEEPS_CLIENT_CALL
),
ASSOCIATION_KEEPS_CLIENT_CALL_1[callSentByClient:CLIENT_CALLS] =
(
     client[C].dos -> ASSOCIATION_KEEPS_CLIENT_CALL
     | client[C].call[CLIENT_CALLS] -> ERROR
     | associate[s:SERVER_THREADS][C] ->
ASSOCIATION_KEEPS_CLIENT_CALL_2[callSentByClient][s]
     | st[SERVER_THREADS].clientCall[CLIENT_CALLS] ->
ASSOCIATION_KEEPS_CLIENT_CALL_1[callSentByClient]
),
ASSOCIATION_KEEPS_CLIENT_CALL_2[callSentByClient:CLIENT_CALLS][associatedServerThread:S
ERVER_THREADS] =
(
     st[s:SERVER_THREADS].clientCall[d:CLIENT_CALLS] -> if (s==associatedServerThread
&& d==callSentByClient) then ASSOCIATION_KEEPS_CLIENT_CALL else if
(associatedServerThread != s) then
ASSOCIATION_KEEPS_CLIENT_CALL_2[callSentByClient][associatedServerThread] else ERROR
     | client[C].dos -> ERROR
     | client[C].call[CLIENT_CALLS] -> ERROR
     | associate[SERVER_THREADS][C] -> ERROR
).
```

```
ASSOCIATION_KEEPS_SERVER_RESPONSE(C=0) =
(
     associate[s:SERVER_THREADS][C] -> ASSOCIATION_KEEPS_SERVER_RESPONSE_1[s]
     | st[SERVER_THREADS].clientResponse[SERVER_RESPONSES] ->
ASSOCIATION_KEEPS_SERVER_RESPONSE
    | client[C].response[SERVER_RESPONSES] -> ERROR

),
ASSOCIATION_KEEPS_SERVER_RESPONSE_1[associatedServerThread:SERVER_THREADS] =
(
     associate[SERVER_THREADS][C] -> ERROR
     | st[s:SERVER_THREADS].clientResponse[r:SERVER_RESPONSES] -> if
(associatedServerThread == s) then ASSOCIATION_KEEPS_SERVER_RESPONSE_2[r] else
ASSOCIATION_KEEPS_SERVER_RESPONSE_1[associatedServerThread]
     | client[C].response[SERVER_RESPONSES] -> ERROR
),
ASSOCIATION_KEEPS_SERVER_RESPONSE_2[serverResponse:SERVER_RESPONSES] =
(
     client[C].response[r:SERVER_RESPONSES] -> if(r==serverResponse) then
ASSOCIATION_KEEPS_SERVER_RESPONSE else ERROR
     | st[SERVER_THREADS].clientResponse[SERVER_RESPONSES] ->
ASSOCIATION_KEEPS_SERVER_RESPONSE_2[serverResponse]
     | associate[s:SERVER_THREADS][C] -> ERROR
).
```

```
property FAIR_ASSOCIATIONS =
(
     client[c0:CLIENTS].call[CLIENT_CALLS] -> FAIR_ASSOCIATIONS[c0]
),
FAIR_ASSOCIATIONS[c0:CLIENTS] =
(
     client[c1:CLIENTS].call[CLIENT_CALLS] -> FAIR_ASSOCIATIONS[c0][c1]
     | client[c0].dos -> FAIR_ASSOCIATIONS
     | associate[SERVER_THREADS][c0] -> FAIR_ASSOCIATIONS
),
FAIR_ASSOCIATIONS[c0:CLIENTS][c1:CLIENTS] =
(
     client[c2:CLIENTS].call[CLIENT_CALLS] -> FAIR_ASSOCIATIONS[c0][c1][c2]
     | client[c0].dos -> FAIR_ASSOCIATIONS[c1]
     | client[c1].dos -> FAIR_ASSOCIATIONS[c0]
     | associate[SERVER_THREADS][c0] -> FAIR_ASSOCIATIONS[c1]
),
FAIR_ASSOCIATIONS[c0:CLIENTS][c1:CLIENTS][c2:CLIENTS] =
```

```
(
    client[c0].dos -> FAIR_ASSOCIATIONS[c1][c2]
    | client[c1].dos -> FAIR_ASSOCIATIONS[c0][c2]
    | client[c2].dos -> FAIR_ASSOCIATIONS[c0][c1]
    | associate[SERVER_THREADS][c0] -> FAIR_ASSOCIATIONS[c1][c2]
).
```

- **Configuración formal:** La configuración formal está dada por $\{client_c; server; conx\}$, con $client_c$ = {`client[c].*`} para `c:CLIENTS`, $server$ = {`st[SERVER_THREADS].*`} y $conx$ = {`associate[SERVER_THREADS][CLIENTS],disassociate[SERVER_THREADS][CLIENTS]`}

- **Sub-estilos:** Cualquier implementación del estilo deberá respetar todas las propiedades exceptuando la propiedad `FAIR_ASSOCIATIONS`. De acuerdo a la especificación presentada existe un único sub-estilo que introduce la restricción impuesta por la propiedad `FAIR_ASSOCIATIONS`. Ésta impone un orden FIFO en la atención de las llamadas.

7.3 Estilo Pipe + Filter

Nombre: Pipe + Filter

Descripción informal:

El estilo Pipe + Filter consiste en la transformación sucesiva de un stream de datos mediante componentes que lo procesan. Los componentes se encuentran conectados entre sí mediante pipes. Los pipes canalizan la salida de un componente hacia la entrada de otro.

Componentes:

- **Sink:** Un sink es un componente que se encuentra asociado a uno o más pipes únicamente a través del rol de salida

- **Producer:** Un producer es un componente que se encuentra asociado a uno o más pipes únicamente a través del rol de entrada

- **Filtro:** Un filtro es un componente que se encuentra asociado a uno o más pipes a través del rol de entrada y uno o más pipes a través del rol de salida. Un filtro se puede pensar como un componente que deriva elementos (tal vez previa transformación de los mismos) entre pipes.

Conectores:

- **Pipe:** Un pipe es un conector con dos roles, el rol de escritura y el rol lectura. A través del rol de escritura se introducen elementos en el pipe. A través del rol de lectura se quitan elementos del pipe. Los elementos dentro del pipe quedan configurados en forma de cola. Cada rol puede estar asociado a un único componente.

Configuraciones legales:

Las configuraciones legales quedan determinadas por los roles definidos por el conector pipe y por las definiciones de los componentes.

Formalización:

- **Propiedades:** Las propiedades planteadas predican sobre el comportamiento de los Pipes (`PIPE_BEHAVIOR`) y presentan diferentes comportamientos posibles para los filtros.

```
property PIPE_BEHAVIOR = EMPTY_PIPE_BEHAVIOR,
EMPTY_PIPE_BEHAVIOR = (in[x:DATA] -> PIPE_BEHAVIOR[x]),
PIPE_BEHAVIOR[d1:DATA] =
(
    in[x:DATA] -> PIPE_BEHAVIOR[x][d1]
    | out[d1] -> PIPE_BEHAVIOR
),
PIPE_BEHAVIOR[d2:DATA][d1:DATA] =
(
    in[x:DATA] -> PIPE_BEHAVIOR[x][d2][d1]
    | out[d1] -> PIPE_BEHAVIOR[d2]
),
PIPE_BEHAVIOR[d3:DATA][d2:DATA][d1:DATA] =
(
    out[d1] -> PIPE_BEHAVIOR[d3][d2]
).
```

```
property FILTER_BUFFER_1_1(S=0,E=1) = FILTER_BUFFER_1_1[S],
FILTER_BUFFER_1_1[bufferCount:S..E] =
(
    when (bufferCount == S) pipeIn.out[DATA] -> FILTER_BUFFER_1_1[bufferCount + 1]
    | when (bufferCount > S && bufferCount < E) pipeIn.out[DATA] ->
FILTER_BUFFER_1_1[bufferCount + 1]
    | when (bufferCount > S && bufferCount < E) pipeOut.in[DATA] ->
FILTER_BUFFER_1_1[bufferCount - 1]
    | when (bufferCount == E) pipeOut.in[DATA] -> FILTER_BUFFER_1_1[bufferCount - 1]
).
```

```
property FILTER_1_1 =
(
    pipeIn.out[DATA] -> pipeOut.in[DATA] -> FILTER_1_1
).
```

```
property FILTER_1_N =
(
    pipeIn.out[DATA] -> pipeOut.in[DATA] -> FILTER_1_N_N
),
FILTER_1_N_N =
(
    pipeOut.in[DATA] -> FILTER_1_N_N
    | pipeIn.out[DATA] -> pipeOut.in[DATA] -> FILTER_1_N_N
).
```

```
property SINK_BEHAVIOR =
```

```
(
    pipeIn.out[DATA] -> SINK_BEHAVIOR
).
```

```
property PRODUCER_BEHAVIOR =
(
    pipeOut.in[DATA] -> PRODUCER_BEHAVIOR
).
```

- **Configuración formal:** No es necesaria configuración formal ya que las restricciones de configuración son suficientes para restringir los lenguajes exhibidos por cada uno de los componentes. Por lo tanto la configuración formal es el conjunto vacio.

- **Sub-estilos:** El estilo Pipes + Filters sea, tal vez, el que más sub-estilos tenga. Se pueden generar incontables sub-estilo a partir de la elección de propiedades para los filtros y al restringir la configuración. Por ejemplo el sub-estilo "1-1" define que todos los filtros deben cumplir la propiedad `FILTER_1_1`

7.4 Estilo Shared Variable

Nombre: Shared Variable

Descripción informal:

El estilo Shared Variable reifica la existencia de una variable compartida que puede ser utilizada por los diferentes componentes de una arquitectura

Componentes:

- **Variable Vault:** El componente variable vault contiene a la variable compartida

- **Reader/Writer:** Cada uno de los componentes reader/writer leen y escriben el valor de la variable compartida

Conectores:

- **Conector Shared Variable:** El conector shared variable recibe las operaciones de escritura y lectura de los reader/writers y la canaliza para que la variable se actualice en forma acorde

Configuraciones legales:

Las configuraciones legales quedan determinadas por los roles definidos por el único conector del estilo (un único Variable vault – múltiples reader/writers asociados a cada

conector)

Formalización:

- **Propiedades:** El estilo define dos propiedades que predican el comportamiento que debe tener la variable frente a las escrituras y lecturas de los reader/writers. Se utiliza el rango `RW` para identificar a los reader/writers del sistema y el rango `DATA` para representar el tipo de la variable

```
property RWS_VARIABLE_BEHAVIOR = RWS_VARIABLE_BEHAVIOR[INITIAL_DATA_VALUE],
RWS_VARIABLE_BEHAVIOR[d:DATA] =
(
    rw[RWS].read[d] -> RWS_VARIABLE_BEHAVIOR[d]
    | rw[RWS].write[w:DATA] -> RWS_VARIABLE_BEHAVIOR[w]
).
```

```
property RW_BEHAVIOR(RW=1) = (rw[RW].read[d:DATA] -> RW_BEHAVIOR | rw[RW].write[d:DATA]
-> RW_BEHAVIOR).
```

- **Configuración formal:** La configuración formal está dada por $\{rw_i\}$, con rw_i = {`rw[i].*`}

- **Sub-estilos:** El estilo es lo suficientemente simple para no contar con especializaciones de ningún tipo

7.5 Estilo Signal

Nombre: Signal

Descripción Informal:

El estilo Signal representa la primitiva de comunicación más básica y se formaliza mediante una acción compartida entre dos componentes conectados y un conector que los conecta. Dada su simplicidad, no aporta propiedades al sistema que lo utilice.

8. Apéndice B – Código Fuente

En este apéndice se lista el código fuente correspondiente a cada uno de los diferentes ejemplos presentados en el trabajo. Para comodidad del lector se utiliza un hipotético operador de pre-proceso `#INCLUDE`, que incluye el contenido completo de un archivo. Nótese que este operador no existe en LTSA y para utilizar los listados aquí presentados, debe reemplazarse manualmente cada aparición del mismo por el contenido del archivo al que hace referencia. Cada ejemplo incluye una introducción en la que se explica brevemente las estrategias de implementación utilizadas y el funcionamiento de los principales procesos.

8.1 NewsService_AnnouncerListener.lts

Introducción

En este ejemplo se utilizan tres procesos principales correspondientes a la agencia de noticias, al conector de *Announcer-Listener* y a las redacciones de noticias. El proceso `NEWS_SERVICE` modela el comportamiento de la agencia de noticias. Este consiste únicamente en la redacción y el anuncio de noticias. El proceso `NEWSPAPER` se utiliza para definir el comportamiento de cada una de las redacciones de noticias. Este comportamiento consiste en la suscripción y desuscripción continua al servicio de noticias. Estando suscriptas, las redacciones esperan los anuncios de noticias de la agencia. El proceso `CONX` corresponde al conector del estilo utilizado. Este se obtiene a partir de la composición de un proceso `CONX_NEWSPAPER_QUEUE` para cada uno de las redacciones de noticias. Cada proceso `CONX_NEWSPAPER_QUEUE` mantiene el estado necesario para lograr en el envió de los eventos a cada redacción, encolándolos de ser necesario (es decir, si la redacción de noticias se encuentra suscripta) o desechándolos en caso contrario (si la redacción de noticias no se encuentra suscripta).

Código fuente

```
// -------------------------------------------------
// | (!) NEWS SERVICE EXAMPLE USING ANNOUNCE-LISTENER |
// -------------------------------------------------

const NEWSPAPER_MIN = 1
const NEWSPAPER_MAX = 3
range NEWSPAPERS = NEWSPAPER_MIN..NEWSPAPER_MAX

const NEWS_MIN = 1
const NEWS_MAX = 2
const NEWS_A = NEWS_MIN
const NEWS_B = NEWS_MAX
range NEWS = NEWS_A..NEWS_B

// ANNOUNCE-LISTENERS SPECIFICATION CONSTS
// ---------------------------------------

const FALSE = 0
const TRUE = 1
range BOOL = FALSE..TRUE

const LMIN = NEWSPAPER_MIN
const LMAX = NEWSPAPER_MAX
range LISTENERS = LMIN..LMAX

const DATA_MIN = NEWS_MIN
const DATA_MAX = NEWS_MAX
```

```
range DATA = DATA_MIN..DATA_MAX

// NEWS_SERVCE
// -----------

NEWS_SERVICE = (newsRedacted[n:NEWS] -> sendNewsAlert[n] -> NEWS_SERVICE).

// NEWSPAPER
// ---------

NEWSPAPER =
(
       subscribe -> NEWSPAPER_SUBSCRIBED
),
NEWSPAPER_SUBSCRIBED =
(
       newsAlertReceived[n:NEWS] -> newsAlertOnScreen[n] -> NEWSPAPER_SUBSCRIBED
       | unsubscribe -> NEWSPAPER
).

|| NEWSPAPER_1 = (newspaper[1]:NEWSPAPER).

|| NEWSPAPER_2 = (newspaper[2]:NEWSPAPER).

|| NEWSPAPER_3 = (newspaper[3]:NEWSPAPER).

// CONNECTOR
// ---------

CONX_NEWSPAPER_QUEUE(N=NEWSPAPER_MIN) =
(
       sendNewsAlert[NEWS] -> CONX_NEWSPAPER_QUEUE
       | newspaper[N].subscribe -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED
),
CONX_NEWSPAPER_QUEUE_SUBSCRIBED =
(
       sendNewsAlert[d:NEWS] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[d]
       | newspaper[N].unsubscribe -> CONX_NEWSPAPER_QUEUE
),
CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1:NEWS] =
(
       sendNewsAlert[d:NEWS] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1][d]
       | newspaper[N].newsAlertReceived[e1] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED
       | newspaper[N].unsubscribe -> CONX_NEWSPAPER_QUEUE
),
CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1:NEWS][e2:NEWS] =
(
       sendNewsAlert[d:NEWS] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1][e2][d]
       | newspaper[N].newsAlertReceived[e1] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e2]
       | newspaper[N].unsubscribe -> CONX_NEWSPAPER_QUEUE
),
CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e1:NEWS][e2:NEWS][e3:NEWS] =
(
       newspaper[N].newsAlertReceived[e1] -> CONX_NEWSPAPER_QUEUE_SUBSCRIBED[e2][e3]
       | newspaper[N].unsubscribe -> CONX_NEWSPAPER_QUEUE
).

|| CONX = (forall[n:NEWSPAPERS] CONX_NEWSPAPER_QUEUE(n)).

// SYSTEM PROPERTIES
// -----------------
property DISPLAY_CORRECT_NEWS_ONSCREEN =
(
       newsAlertReceived[n:NEWS] -> newsAlertOnScreen[n] -> DISPLAY_CORRECT_NEWS_ONSCREEN
).

// ARCHITECTURE
// ------------

CANNONICAL_STYLE_TRANSLATION_LISTENER(L=LMIN) =
(
```

```
        newspaper[L].newsAlertReceived[n:NEWS] -> listener[L].event[n]
         -> CANNONICAL_STYLE_TRANSLATION_LISTENER
        |newspaper[L].subscribe -> listener[L].subscribe
         -> CANNONICAL_STYLE_TRANSLATION_LISTENER
        | newspaper[L].unsubscribe -> listener[L].unsubscribe
         -> CANNONICAL_STYLE_TRANSLATION_LISTENER
).

CANNONICAL_STYLE_TRANSLATION_ANNOUNCE =
(
        sendNewsAlert[n:NEWS] -> announce[n] -> CANNONICAL_STYLE_TRANSLATION_ANNOUNCE
).

|| ARQ =
(
        NEWS_SERVICE
        || CONX
        || NEWSPAPER_1
        || NEWSPAPER_2
        || NEWSPAPER_3
).

// CHECKED ARCHITECTURE
// --------------------

|| CONX_AS_ANNOUNCE_LISTENER_SPECIFICATION =
(
        CONX
        || CANNONICAL_STYLE_TRANSLATION_ANNOUNCE
        || forall[l:LISTENERS] CANNONICAL_STYLE_TRANSLATION_LISTENER(l)
)
<< {
        listener[LISTENERS].subscribe, listener[LISTENERS].unsubscribe,
        listener[LISTENERS].event[DATA], announce[DATA]
}.

|| NEWS_SERVICE_AS_ANNOUNCE_LISTENER_SPECIFICATION  =
        (NEWS_SERVICE || CANNONICAL_STYLE_TRANSLATION_ANNOUNCE) << {announce[DATA]}.

|| NEWSPAPER_1_AS_ANNOUNCE_LISTENER_SPECIFICATION =
        (NEWSPAPER_1 || CANNONICAL_STYLE_TRANSLATION_LISTENER(1))
        << {listener[1].subscribe, listener[1].unsubscribe, listener[1].event[DATA]}.

|| NEWSPAPER_2_AS_ANNOUNCE_LISTENER_SPECIFICATION =
        (NEWSPAPER_2 || CANNONICAL_STYLE_TRANSLATION_LISTENER(2))
        << {listener[2].subscribe, listener[2].unsubscribe, listener[2].event[DATA]}.


|| NEWSPAPER_3_AS_ANNOUNCE_LISTENER_SPECIFICATION =
        (NEWSPAPER_3 || CANNONICAL_STYLE_TRANSLATION_LISTENER(3))
        << {listener[3].subscribe, listener[3].unsubscribe, listener[3].event[DATA]}.

|| ARQ_AS_ANNOUNCE_LISTENER_SPECIFICATION =
(
        NEWS_SERVICE_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || CONX_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || NEWSPAPER_1_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || NEWSPAPER_2_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || NEWSPAPER_3_AS_ANNOUNCE_LISTENER_SPECIFICATION
).

|| ARQ_IS_ANNOUNCE_LISTENER =
(
        ARQ_AS_ANNOUNCE_LISTENER_SPECIFICATION
        || forall[l:LISTENERS] LISTENER_BEHAVIOR(l)
        || forall[l:LISTENERS] NO_MISSING_EVENTS(l,3)
        || forall[l:LISTENERS] CORRECT_EVENTS(l)
        || ANNOUNCER_BEHAVIOR
).

|| CHECKED_ARQ =
```

```
(
        ARQ
        || forall[n:NEWSPAPERS] newspaper[n]:DISPLAY_CORRECT_NEWS_ONSCREEN
).

// -------------------------------------
// LISTENER-ANNOUNCER STYLE SPECIFICATION
// -------------------------------------

#INCLUDE “ListenerAnnouncer.lts”
```

8.2 AnnouncerListener.lts

Introducción

La descripción de cada uno de los procesos/propiedades detallados en este listado puede encontrarse en el catálogo de estilos del Apéndice A.

Código fuente

```
// -------------------------------------
// LISTENER-ANNOUNCER STYLE SPECIFICATION
// -------------------------------------

/*

 ROLES:
        1 - LISTENER (1..*) L{} = announceEvent
        2 - ANNOUNCER (1) L() = subscribe, unsubscribe, listenEvent, FLUENT
LISTENER_SUBSCRIBED

CONNECTORS:
        1 - CS CONX (1)
                -> Role Listeners (1..*) - LISTEN
                -> Role Announcer (1) - ANNOUNCER

CONFIGURATION:
        - Restrictions: none beyond those imposed by the connectors
*/

fluent FLUENT_LISTENER_SUBSCRIBED[l:LISTENERS] =
                                    <{listener[l].subscribe},{listener[l].unsubscribe}>

property ANNOUNCER_BEHAVIOR = (announce[DATA] -> ANNOUNCER_BEHAVIOR).

property LISTENER_BEHAVIOR(L=LMIN) =
(
        listener[L].subscribe -> LISTENER_BEHAVIOR_SUBSCRIBED
),
LISTENER_BEHAVIOR_SUBSCRIBED =
(
        listener[L].event[DATA] -> LISTENER_BEHAVIOR_SUBSCRIBED
        | listener[L].unsubscribe -> LISTENER_BEHAVIOR
).

property CORRECT_EVENTS(L=LMIN) =
(
        listener[L].subscribe -> CORRECT_EVENTS_SUBSCRIBED
        | announce[DATA] -> CORRECT_EVENTS
),
CORRECT_EVENTS_SUBSCRIBED =
(
        listener[L].unsubscribe -> CORRECT_EVENTS
        | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[d]
),

CORRECT_EVENTS_SUBSCRIBED[e1:DATA] =
(
        listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED
        | listener[L].unsubscribe -> CORRECT_EVENTS
        | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[e1][d]
),
CORRECT_EVENTS_SUBSCRIBED[e1:DATA][e2:DATA] =
(
        listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED[e2]
        | listener[L].unsubscribe -> CORRECT_EVENTS
        | announce[d:DATA] -> CORRECT_EVENTS_SUBSCRIBED[e1][e2][d]
),
CORRECT_EVENTS_SUBSCRIBED[e1:DATA][e2:DATA][e3:DATA] =
```

```
(
        listener[L].event[e1] -> CORRECT_EVENTS_SUBSCRIBED[e2][e3]
        | listener[L].unsubscribe -> CORRECT_EVENTS
).
```

8.3 NewsService_ClientServer.lts

Introducción

En este ejemplo se utilizan tres procesos principales correspondientes a la agencia de noticias, al conector de *Client-Server* y a las redacciones de noticias. El proceso `NEWSPAPER` se utiliza para definir el comportamiento de cada una de las redacciones de noticias. Este comportamiento consiste en la consulta continua por nuevos eventos mediante llamadas `QUERY_NEWS_CALL` al servidor. Ante este llamado, el servidor puede responder `NO_NEWS` (no hay noticias nuevas) o bien un elemento del rango `NEWS` (es decir, retorna la nueva noticia). La desuscripción al servicio se realiza mediante una llamada `CANCEL_NEWS_SUBSCRIPTION` al servidor. La suscripción se realiza automáticamente al consultar por nuevos eventos mediante una llamada `QUERY_NEWS_CALL`. El proceso `NEWS_SERVICE` describe el comportamiento de la agencia de noticias. Este resulta de la composición de los procesos `SERVER_NEWS_GENERATION`, `SERVER_THREAD` y `SERVER_NEWS_QUEUES`. El primero consiste en la creación de nuevas noticias y su posterior "encolado" en las colas de noticias/eventos pendientes de cada uno de los clientes que se encuentren suscriptos. El segundo proceso representa a los threads de ejecución del servidor; su función es recibir los pedidos de los clientes y generar las respuestas adecuadas para los mismos. El tercer proceso define el comportamiento de las colas de noticias/eventos pendientes para los clientes. El proceso `CONX` corresponde al conector del estilo utilizado. Este se obtiene a partir de la composición de un proceso `CS_CONX_ST_STATUS` para cada uno de los threads del servidor y un proceso `CS_CONX_CLIENT` para cada uno de los clientes. Los primeros llevan registro del estado de cada thread del servidor (libre u ocupado). Los segundos relacionan cada llamada de los clientes con un thread libre. Si no existieran threads libres al recibir la llamada, se niega la conexión al servidor.

Código fuente

```
// -----------------------------------------------
// | (!) NEWS SERVICE EXAMPLE USING CLIENT-SERVER |
// -----------------------------------------------

const NEWSPAPER_MIN = 1
const NEWSPAPER_MAX = 3
range NEWSPAPERS = NEWSPAPER_MIN..NEWSPAPER_MAX

const NEWS_MIN = 0
const NEWS_MAX = 1
range NEWS = NEWS_MIN..NEWS_MAX

const QUERY_NEWS_CALL = 1
const CANCEL_NEWS_SUBSCRIPTION = 2

const NO_NEWS = NEWS_MIN - 1

// CS SPECIFICATION CONSTS
// -----------------------

const CLIENT_MIN = NEWSPAPER_MIN
const CLIENT_MAX = NEWSPAPER_MAX
range CLIENTS = CLIENT_MIN..CLIENT_MAX

const CALL_MIN = QUERY_NEWS_CALL
const CALL_MAX = CANCEL_NEWS_SUBSCRIPTION
range CLIENT_CALLS = CALL_MIN..CALL_MAX
```

```
const RESPONSE_MIN = NO_NEWS
const RESPONSE_MAX = NEWS_MAX
range SERVER_RESPONSES = RESPONSE_MIN..RESPONSE_MAX

range SERVER_THREADS = 0..1
range SERVER_THREADS_AND_BUSY = -1..1
const SERVER_THREAD_BUSY = -1

// ANNOUNCE-LISTENERS SPECIFICATION CONSTS
// ---------------------------------------

const FALSE = 0
const TRUE = 1
range BOOL = FALSE..TRUE

const LMIN = NEWSPAPER_MIN
const LMAX = NEWSPAPER_MAX
range LISTENERS = LMIN..LMAX

const DATA_MIN = NEWS_MIN
const DATA_MAX = NEWS_MAX
range DATA = DATA_MIN..DATA_MAX

// NEWSPAPER
// ---------

NEWSPAPER =
(
        call[QUERY_NEWS_CALL] ->
        (
                response[NO_NEWS] -> NEWSPAPER_SUBSCRIBED
                | response[n:NEWS] ->  newsAlertOnScreen[n] -> NEWSPAPER_SUBSCRIBED
                | dos -> NEWSPAPER
        )
),
NEWSPAPER_SUBSCRIBED =
(
        call[QUERY_NEWS_CALL] ->
        (
                response[NO_NEWS] -> NEWSPAPER_SUBSCRIBED
                | response[n:NEWS] ->  newsAlertOnScreen[n] -> NEWSPAPER_SUBSCRIBED
                | dos -> NEWSPAPER_SUBSCRIBED
        )
        | call[CANCEL_NEWS_SUBSCRIPTION] ->
        (
                response[SERVER_RESPONSES] -> NEWSPAPER
                | dos -> NEWSPAPER_SUBSCRIBED
        )
).


|| NEWSPAPER_1 = (client[1]:NEWSPAPER)
        / {newspaper[1].newsAlertOnScreen[n:NEWS] / client[1].newsAlertOnScreen[n]}.

|| NEWSPAPER_2 = (client[2]:NEWSPAPER)
        / {newspaper[2].newsAlertOnScreen[n:NEWS] / client[2].newsAlertOnScreen[n]}.

|| NEWSPAPER_3 = (client[3]:NEWSPAPER)
        / {newspaper[3].newsAlertOnScreen[n:NEWS] /client[3].newsAlertOnScreen[n]}.

// NEWS_SERVICE
// ------------

SERVER_THREAD(T=0) =
(
        st[T].call[c:CLIENTS][CANCEL_NEWS_SUBSCRIPTION]
         -> st[T].response[c][SERVER_RESPONSES]
         -> SERVER_THREAD
        | st[T].call[c:CLIENTS][QUERY_NEWS_CALL]
         -> st[T].response[c][SERVER_RESPONSES]
```

```
         -> SERVER_THREAD
).

SERVER_NEWS_GENERATION =
(
        newsRedacted[n:NEWS] -> enqueueNewsAlert[NEWS] -> SERVER_NEWS_GENERATION
).

SERVER_NEWS_QUEUES(C=CLIENT_MIN) =
(
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION]
         -> st[SERVER_THREADS].response[C][NO_NEWS] -> SERVER_NEWS_QUEUES
        | st[SERVER_THREADS].call[C][QUERY_NEWS_CALL]
         -> st[SERVER_THREADS].response[C][NO_NEWS] -> SERVER_NEWS_QUEUES_SUBSCRIBED
        | enqueueNewsAlert[n:NEWS] -> SERVER_NEWS_QUEUES
),
SERVER_NEWS_QUEUES_SUBSCRIBED =
(
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION]
         -> st[SERVER_THREADS].response[C][NO_NEWS] -> SERVER_NEWS_QUEUES
        | st[SERVER_THREADS].call[C][QUERY_NEWS_CALL]
         -> st[SERVER_THREADS].response[C][NO_NEWS] -> SERVER_NEWS_QUEUES_SUBSCRIBED
        | enqueueNewsAlert[n:NEWS] -> SERVER_NEWS_QUEUES[n]
),
SERVER_NEWS_QUEUES[n1:NEWS] =
(
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION]
         -> st[SERVER_THREADS].response[C][NO_NEWS] -> SERVER_NEWS_QUEUES
        | st[SERVER_THREADS].call[C][QUERY_NEWS_CALL]
         -> st[SERVER_THREADS].response[C][n1] -> SERVER_NEWS_QUEUES_SUBSCRIBED
        | enqueueNewsAlert[n:NEWS] -> SERVER_NEWS_QUEUES[n1][n]
),
SERVER_NEWS_QUEUES[n1:NEWS][n2:NEWS] =
(
        st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION]
         -> st[SERVER_THREADS].response[C][NO_NEWS]
         -> SERVER_NEWS_QUEUES
        | st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] -> st[SERVER_THREADS].response[C][n1]
         -> SERVER_NEWS_QUEUES[n2]
).

|| NEWS_SERVICE =
(
        SERVER_NEWS_GENERATION
        || forall[t:SERVER_THREADS] SERVER_THREAD(t)
        || forall[c:CLIENTS] SERVER_NEWS_QUEUES(c)
).

// CONNECTOR
// ---------
CS_CONX_ST_STATUS(T=0) = CS_CONX_ST_STATUS_FREE,
CS_CONX_ST_STATUS_FREE =
(
        st[T].call[c:CLIENTS][CLIENT_CALLS] -> CS_CONX_ST_STATUS_BUSY[c]
),
CS_CONX_ST_STATUS_BUSY[c:CLIENTS] =
(
        st[SERVER_THREAD_BUSY].call[CLIENTS][CLIENT_CALLS] -> CS_CONX_ST_STATUS_BUSY[c]
        | st[T].response[c][SERVER_RESPONSES] -> CS_CONX_ST_STATUS_FREE
).

CS_CONX_CLIENT(C=0) =
(
        client[C].call[call:CLIENT_CALLS] ->
        (
                st[s:SERVER_THREADS].call[C][call] -> st[s].response[C][r:SERVER_RESPONSES]
                 -> client[C].response[r]  -> CS_CONX_CLIENT
                | st[SERVER_THREAD_BUSY].call[C][call] -> CS_CONX_CLIENT
        )
).
```

```
|| CS_CONX =
(
        forall[t:SERVER_THREADS] CS_CONX_ST_STATUS(t)
        || forall[c:CLIENTS] CS_CONX_CLIENT(c)
)
/ {client[c:CLIENTS].dos / st[SERVER_THREAD_BUSY].call[c][CLIENT_CALLS]}.

// SYSTEM PROPERTIES
// -----------------
property DISPLAY_CORRECT_NEWS_ONSCREEN =
(
        newsAlertReceived[n:NEWS] -> newsAlertOnScreen[n] -> DISPLAY_CORRECT_NEWS_ONSCREEN
).

// ARCHITECTURE
// ------------

CANNONICAL_STYLE_TRANSLATION_LISTENER(L=LMIN) =
(
        newspaper[L].newsAlertReceived[n:NEWS]
         -> listener[L].event[n]
         -> CANNONICAL_STYLE_TRANSLATION_LISTENER
        | newspaper[L].subscribe
         -> listener[L].subscribe
         -> CANNONICAL_STYLE_TRANSLATION_LISTENER
        | newspaper[L].unsubscribe
         -> listener[L].unsubscribe
         -> CANNONICAL_STYLE_TRANSLATION_LISTENER
).

CANNONICAL_STYLE_TRANSLATION_ANNOUNCE =
(
        sendNewsAlert[n:NEWS] -> announce[n] -> CANNONICAL_STYLE_TRANSLATION_ANNOUNCE
).

REFINEMENT_TRANSLATION =
(
        client[c:CLIENTS].response[n:NEWS]
         -> newspaper[c].newsAlertReceived[n] -> REFINEMENT_TRANSLATION
        | st[SERVER_THREADS].call[c:CLIENTS][CANCEL_NEWS_SUBSCRIPTION]
         -> newspaper[c].unsubscribe -> REFINEMENT_TRANSLATION
        | enqueueNewsAlert[n:NEWS] -> sendNewsAlert[n]  -> REFINEMENT_TRANSLATION
).

REFINEMENT_TRANSLATION_SUBSCRIBE(C=CLIENT_MIN) =
(
        st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] -> newspaper[C].subscribe
         -> REFINEMENT_TRANSLATION_SUBSCRIBE_S
        | st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION]
         -> REFINEMENT_TRANSLATION_SUBSCRIBE
),
REFINEMENT_TRANSLATION_SUBSCRIBE_S =
(
        st[SERVER_THREADS].call[C][QUERY_NEWS_CALL] -> REFINEMENT_TRANSLATION_SUBSCRIBE_S
        | st[SERVER_THREADS].call[C][CANCEL_NEWS_SUBSCRIPTION]
         -> REFINEMENT_TRANSLATION_SUBSCRIBE
).

|| ARQREF =
(
        CS_CONX
        || NEWSPAPER_1
        || NEWSPAPER_2
        || NEWSPAPER_3
        || NEWS_SERVICE
).

// CHECKED ARCHITECTURE
// --------------------

CANNONICAL_CS_TRANSLATION_ASSOCIATE =
```

```
(
        st[t:SERVER_THREADS].call[c:CLIENTS][call:CLIENT_CALLS] -> associate[t][c]
        -> st[t].clientCall[call] -> CANNONICAL_CS_TRANSLATION_ASSOCIATE
).

CANNONICAL_CS_TRANSLATION_DISASSOCIATE =
(
        st[t:SERVER_THREADS].response[c:CLIENTS][response:SERVER_RESPONSES]
        -> st[t].clientResponse[response] -> disassociate[t][c]
        -> CANNONICAL_CS_TRANSLATION_DISASSOCIATE
).

|| NEWS_SERVICE_AS_CS_SPECIFICATION =
(
        NEWS_SERVICE
        || CANNONICAL_CS_TRANSLATION_ASSOCIATE
        || CANNONICAL_CS_TRANSLATION_DISASSOCIATE
)  << {
        associate[t:SERVER_THREADS][c:CLIENTS],
        disassociate[t:SERVER_THREADS][c:CLIENTS],
        st[SERVER_THREADS].clientCall[CLIENT_CALLS],
        st[SERVER_THREADS].clientResponse[SERVER_RESPONSES]
}.

|| CS_CONX_AS_CS_SPECIFICATION =
(
        CS_CONX
        || CANNONICAL_CS_TRANSLATION_ASSOCIATE
        || CANNONICAL_CS_TRANSLATION_DISASSOCIATE
)  << {
        associate[t:SERVER_THREADS][c:CLIENTS],
        disassociate[t:SERVER_THREADS][c:CLIENTS],
        st[SERVER_THREADS].clientCall[CLIENT_CALLS],
        st[SERVER_THREADS].clientResponse[SERVER_RESPONSES]
}.

|| ARQREF_AS_CS_SPECIFICATION =
(
        CS_CONX_AS_CS_SPECIFICATION
        || NEWSPAPER_1
        || NEWSPAPER_2
        || NEWSPAPER_3
        || NEWS_SERVICE_AS_CS_SPECIFICATION
).

|| ARQREF_IS_CS =
(
        ARQREF_AS_CS_SPECIFICATION
        || forall[c:CLIENTS] client[c]:CLIENT_BEHAVIOR
        || forall[t:SERVER_THREADS] SERVER_THREAD_BEHAVIOR(t)
        || forall[t:SERVER_THREADS] SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR(t)
        || forall[c:CLIENTS] ASSOCIATION_KEEPS_CLIENT_CALL(c)
        || forall[c:CLIENTS] ASSOCIATION_KEEPS_SERVER_RESPONSE(c)
).

|| ARQREF_AS_ABSTRACT_DESCRIPCTION_LANGUAGE =
(
        ARQREF
        || REFINEMENT_TRANSLATION
        || forall[c:CLIENTS] REFINEMENT_TRANSLATION_SUBSCRIBE(c)
) << {
        newspaper[NEWSPAPERS].unsubscribe,
        newspaper[NEWSPAPERS].subscribe,
        newspaper[NEWSPAPERS].newsAlertReceived[n:NEWS],
        sendNewsAlert[n:NEWS]
}.

|| ARQREF_AS_ANNOUNCER_LISTENER_SPECIFICATION =
(
        ARQREF_AS_ABSTRACT_DESCRIPCTION_LANGUAGE
        || forall[l:LISTENERS] CANNONICAL_STYLE_TRANSLATION_LISTENER(l)
```

```
        || CANNONICAL_STYLE_TRANSLATION_ANNOUNCE
) << {
        listener[LISTENERS].unsubscribe,
        listener[LISTENERS].subscribe,
        listener[LISTENERS].event[DATA],
        announce[DATA]
}.

|| ARQREF_IS_REFINEMENT_SYSTEM_PROPS =
(
        ARQREF_AS_ABSTRACT_DESCRIPCTION_LANGUAGE
        || forall[n:NEWSPAPERS] newspaper[n]:DISPLAY_CORRECT_NEWS_ONSCREEN
).

|| ARQREF_IS_REFINEMENT =
(
        ARQREF_AS_ANNOUNCER_LISTENER_SPECIFICATION
        || forall[l:LISTENERS] listener[l]:LISTENER_BEHAVIOR
        || forall[l:LISTENERS] NO_MISSING_EVENTS(l,3)
        || forall[l:LISTENERS] CORRECT_EVENTS(l)
        || ANNOUNCER_BEHAVIOR
        || forall[n:NEWSPAPERS] newspaper[n]:DISPLAY_CORRECT_NEWS_ONSCREEN
).

// ---------------------------------------
// LISTENER-ANNOUNCER STYLE SPECIFICATION
// ---------------------------------------

#INCLUDE “ListenerAnnouncer.lts”

// ---------------------------------
// CLIENT-SERVER STYLE SPECIFICATION
// ---------------------------------

#INCLUDE “ClientServer.lts”
```

8.4 ClientServer.lts

Introducción

La descripción de cada uno de los procesos/propiedades detallados en este listado puede encontrarse en el catálogo de estilos del Apéndice A.

Código fuente

```
// --------------------------------
// CLIENT-SERVER STYLE SPECIFICATION
// --------------------------------

/*

 ROLES:
       1 - CLIENT (1..*)
       2 - SERVER (1)

 CONNECTORS:
       1 - CS CONX (1)
              -> Role Client (1..*) - CLIENT
              -> Role Server (1) - SERVER

 CONFIGURATION:
       - Restrictions: none beyond those imposed by the connectors

*/

// NOTES:
// st stands for server thread

property CLIENT_BEHAVIOR =
(
       call[CLIENT_CALLS] ->
       (
              response[SERVER_RESPONSES] -> CLIENT_BEHAVIOR
              | dos -> CLIENT_BEHAVIOR
       )
).

property SERVER_THREAD_BEHAVIOR(T=0) =
(
       st[T].clientCall[CLIENT_CALLS]
        -> st[T].clientResponse[SERVER_RESPONSES]
        -> SERVER_THREAD_BEHAVIOR
).


property SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR(T=0) =
(
       associate[T][c:CLIENTS] -> st[T].clientCall[CLIENT_CALLS]
       -> st[T].clientResponse[SERVER_RESPONSES] -> disassociate[T][c]
       -> SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR
).

fluent SERVER_THREAD_BUSY[t:SERVER_THREADS] =
                                    <{associate[t][CLIENTS]},{disassociate[t][CLIENTS]}>

assert ASSOCIATE_NOT_BUSY_THREADS =
forall[t:SERVER_THREADS] ([](X(associate[t][CLIENTS]) -> !SERVER_THREAD_BUSY[t]))

assert DOS_ONLY_IF_ALL_TRHEADS_ARE_BUSY =
forall[c:CLIENTS] ([](X(client[c].dos) -> forall[t:SERVER_THREADS] SERVER_THREAD_BUSY[t]))


ASSOCIATION_KEEPS_CLIENT_CALL(C=0) =
(
```

```
        client[C].call[d:CLIENT_CALLS] -> ASSOCIATION_KEEPS_CLIENT_CALL_1[d]
        | client[C].dos -> ERROR
        | associate[s:SERVER_THREADS][C] -> ERROR
        | st[SERVER_THREADS].clientCall[CLIENT_CALLS] -> ASSOCIATION_KEEPS_CLIENT_CALL
),
ASSOCIATION_KEEPS_CLIENT_CALL_1[callSentByClient:CLIENT_CALLS] =
(
        client[C].dos -> ASSOCIATION_KEEPS_CLIENT_CALL
        | client[C].call[CLIENT_CALLS] -> ERROR
        | associate[s:SERVER_THREADS][C]
         -> ASSOCIATION_KEEPS_CLIENT_CALL_2[callSentByClient][s]
        | st[SERVER_THREADS].clientCall[CLIENT_CALLS]
         -> ASSOCIATION_KEEPS_CLIENT_CALL_1[callSentByClient]
),
ASSOCIATION_KEEPS_CLIENT_CALL_2[callSentByClient:CLIENT_CALLS][associatedServerThread:SERVER
_THREADS] =
(
        st[s:SERVER_THREADS].clientCall[d:CLIENT_CALLS]
        -> if (s==associatedServerThread && d==callSentByClient) then
                ASSOCIATION_KEEPS_CLIENT_CALL
        else if (associatedServerThread != s) then
                ASSOCIATION_KEEPS_CLIENT_CALL_2[callSentByClient][associatedServerThread]
        else
                ERROR
        | client[C].dos -> ERROR
        | client[C].call[CLIENT_CALLS] -> ERROR
        | associate[SERVER_THREADS][C] -> ERROR
).

ASSOCIATION_KEEPS_SERVER_RESPONSE(C=0) =
(
        associate[s:SERVER_THREADS][C] -> ASSOCIATION_KEEPS_SERVER_RESPONSE_1[s]
        | st[SERVER_THREADS].clientResponse[SERVER_RESPONSES] ->
ASSOCIATION_KEEPS_SERVER_RESPONSE
        | client[C].response[SERVER_RESPONSES] -> ERROR

),
ASSOCIATION_KEEPS_SERVER_RESPONSE_1[associatedServerThread:SERVER_THREADS] =
(
        associate[SERVER_THREADS][C] -> ERROR
        | st[s:SERVER_THREADS].clientResponse[r:SERVER_RESPONSES]
         -> if (associatedServerThread == s) then
                                          ASSOCIATION_KEEPS_SERVER_RESPONSE_2[r]
                                  else
        ASSOCIATION_KEEPS_SERVER_RESPONSE_1[associatedServerThread]
        | client[C].response[SERVER_RESPONSES] -> ERROR
),
ASSOCIATION_KEEPS_SERVER_RESPONSE_2[serverResponse:SERVER_RESPONSES] =
(
        client[C].response[r:SERVER_RESPONSES]
         -> if(r==serverResponse) then ASSOCIATION_KEEPS_SERVER_RESPONSE else ERROR
        | st[SERVER_THREADS].clientResponse[SERVER_RESPONSES]
         -> ASSOCIATION_KEEPS_SERVER_RESPONSE_2[serverResponse]
        | associate[s:SERVER_THREADS][C] -> ERROR
).

property FAIR_ASSOCIATIONS =
(
        client[c0:CLIENTS].call[CLIENT_CALLS] -> FAIR_ASSOCIATIONS[c0]
),
FAIR_ASSOCIATIONS[c0:CLIENTS] =
(
        client[c1:CLIENTS].call[CLIENT_CALLS] -> FAIR_ASSOCIATIONS[c0][c1]
        | client[c0].dos -> FAIR_ASSOCIATIONS
        | associate[SERVER_THREADS][c0] -> FAIR_ASSOCIATIONS
),
FAIR_ASSOCIATIONS[c0:CLIENTS][c1:CLIENTS] =
(
        client[c2:CLIENTS].call[CLIENT_CALLS] -> FAIR_ASSOCIATIONS[c0][c1][c2]
        | client[c0].dos -> FAIR_ASSOCIATIONS[c1]
        | client[c1].dos -> FAIR_ASSOCIATIONS[c0]
```

```
        | associate[SERVER_THREADS][c0] -> FAIR_ASSOCIATIONS[c1]
),

FAIR_ASSOCIATIONS[c0:CLIENTS][c1:CLIENTS][c2:CLIENTS] =
(
        client[c0].dos -> FAIR_ASSOCIATIONS[c1][c2]
        | client[c1].dos -> FAIR_ASSOCIATIONS[c0][c2]
        | client[c2].dos -> FAIR_ASSOCIATIONS[c0][c1]
        | associate[SERVER_THREADS][c0] -> FAIR_ASSOCIATIONS[c1][c2]
).
```

8.5 MonteCarlo_PipeFilter.lts

Introducción

En este ejemplo se utilizan cuatro procesos principales correspondientes a los pipes, a las fases de simulación y a los componentes de lectura y escritura de proyectos. El comportamiento de los pipes es el esperable y se encuentra modelado mediante el proceso `PIPE`. El componente de lectura del proyecto se encuentra representado por el proceso `PROJECT_READER`, este lee en forma progesiva los datos del proyecto (`readTask`) y los ingresa al pipeline. El componente de escritura del proyecto se encuentra representado por el proceso `PROJECT_WRITER`, este extrae en forma progresiva los datos del proyecto del final del pipeline y los persiste (`writeTaskRisk`). El comportamiento de las fases de simulación se encuentra definido por el proceso `SIMULATION_PHASE`: se extrae un dato de un pipe, se opera sobre el misma y luego se ingresa los resultados en otro pipe.

Código fuente

```
// ------------------------------------------------
// MONTECARLO SIMULATION EXAMPLE USING PIPE + FILTER
// ------------------------------------------------

const TASK_RISK_MIN = 0
const TASK_RISK_MAX = 1
range TASK_RISK = TASK_RISK_MIN..TASK_RISK_MAX

// PIPE + FILTER SPECIFICATION CONSTS
// ---------------------------------
const DATA_MIN = TASK_RISK_MIN
const DATA_MAX = TASK_RISK_MAX
range DATA = DATA_MIN..DATA_MAX

const SIMULATION_COUNT = 2

// SYSTEM PROPERTIES
// -----------------
property SIMULATION_PHASE_WORK =
(
        out[d:TASK_RISK] -> if(d == TASK_RISK_MAX) then
                                        (in[TASK_RISK_MIN] -> SIMULATION_PHASE_WORK)
                                else
                                        (in[d + 1]  -> SIMULATION_PHASE_WORK)
).

PUSH_PROJECT_TO_PIPELINE = (readTask[d:TASK_RISK] -> pipe[1].in[d] ->
PUSH_PROJECT_TO_PIPELINE).

PULL_PROJECT_FROM_PIPELINE = (pipe[3].out[d:TASK_RISK] -> writeTaskRisk[d] ->
PULL_PROJECT_FROM_PIPELINE).

// PIPE
// ----
PIPE = EMPTY_PIPE,
EMPTY_PIPE = (in[x:TASK_RISK] -> PIPE[x]),
PIPE[d1:TASK_RISK] = (in[x:TASK_RISK] -> PIPE[x][d1] | out[d1] -> PIPE),
PIPE[d2:TASK_RISK][d1:TASK_RISK] = (in[x:TASK_RISK] -> PIPE[x][d2][d1] | out[d1] ->
PIPE[d2]),
PIPE[d3:TASK_RISK][d2:TASK_RISK][d1:TASK_RISK] = (out[d1] -> PIPE[d3][d2]).

// PROJECT READER
// --------------
PROJECT_READER = (readTask[d:TASK_RISK] -> in[d] -> PROJECT_READER).

// PROJECT WRITER
```

```
// --------------
PROJECT_WRITER = (out[d:TASK_RISK] -> writeTaskRisk[d] -> PROJECT_WRITER).

// SIMULATION PHASES
// -----------------
SIMULATION_PHASE =
(
       out[d:DATA] -> if (d == TASK_RISK_MAX) then
                                          (in[TASK_RISK_MIN] -> SIMULATION_PHASE)
                                   else
                                          (in[d + 1] -> SIMULATION_PHASE)
).

|| SIMULATION_PHASE_1 = SIMULATION_PHASE
       / { pipe[1].out[d:TASK_RISK] / out[d], pipe[2].in[d:TASK_RISK] / in[d]}.

|| SIMULATION_PHASE_2 = SIMULATION_PHASE
       / { pipe[2].out[d:TASK_RISK] / out[d], pipe[3].in[d:TASK_RISK] / in[d]}.

// ARCHITECTURE
// ------------
|| ARQ =
(
       PROJECT_READER / {pipe[1].in[d:TASK_RISK] / in[d]}
       || PROJECT_WRITER / {pipe[3].out[d:TASK_RISK] / out[d]}
       || SIMULATION_PHASE_1
       || SIMULATION_PHASE_2
       || pipe[1]::PIPE
       || pipe[2]::PIPE
       || pipe[3]::PIPE
).

// CHECKED ARCHITECTURE
// --------------------
|| CHECKED_ARQ =
(
       ARQ
       || pipe[1]::PIPE_BEHAVIOR
       || pipe[2]::PIPE_BEHAVIOR
       || pipe[3]::PIPE_BEHAVIOR
       || PRODUCER_BEHAVIOR
         / { pipe[1].in[d:DATA] / pipeOut.in[d]}
       || SINK_BEHAVIOR
         / { pipe[3].out[d:DATA] / pipeIn.out[d]}
       || FILTER_1_1
         / { pipe[1].out[d:DATA] / pipeIn.out[d], pipe[2].in[d:DATA] / pipeOut.in[d]}
       || FILTER_1_1
         / { pipe[2].out[d:DATA] / pipeIn.out[d], pipe[3].in[d:DATA] / pipeOut.in[d]}
       || FILTER_BUFFER_1_1(0,6)
         / { pipe[1].out[d:DATA] / pipeIn.out[d], pipe[3].in[d:DATA] / pipeOut.in[d]}
       || SIMULATION_PHASE_WORK
         / { pipe[1].out[d:TASK_RISK] / out[d], pipe[2].in[d:TASK_RISK] / in[d]}
       || SIMULATION_PHASE_WORK
        / { pipe[2].out[d:TASK_RISK] / out[d], pipe[3].in[d:TASK_RISK] / in[d]}
       || PUSH_PROJECT_TO_PIPELINE
       || PULL_PROJECT_FROM_PIPELINE
).

// ---------------------------------
// PIPE + FILTER STYLE SPECIFICATION
// ---------------------------------

#INCLUDE “PipeFilter.lts”
```

8.6 PipeFilter.lts

Introducción

La descripción de cada uno de los procesos/propiedades detallados en este listado puede encontrarse en el catálogo de estilos del Apéndice A.

Código fuente

```
// ---------------------------------
// PIPE + FILTER STYLE SPECIFICATION
// ---------------------------------

/*
 ROLES:
       1 - FILTERS (0..*) [ PRODUCERS / SINKS / FILTER ]

 CONNECTORS:
       1 - PIPE (1)
              -> Role In (1..*) - FILTER
              -> Role Out (1..*) - FILTER

 CONFIGURATION:
       - Restrictions: none beyond those imposed by the connectors
*/

property PIPE_BEHAVIOR = EMPTY_PIPE_BEHAVIOR,
EMPTY_PIPE_BEHAVIOR = (in[x:DATA] -> PIPE_BEHAVIOR[x]),
PIPE_BEHAVIOR[d1:DATA] =
(
       in[x:DATA] -> PIPE_BEHAVIOR[x][d1]
       | out[d1] -> PIPE_BEHAVIOR
),
PIPE_BEHAVIOR[d2:DATA][d1:DATA] =
(
       in[x:DATA] -> PIPE_BEHAVIOR[x][d2][d1]
       | out[d1] -> PIPE_BEHAVIOR[d2]
),
PIPE_BEHAVIOR[d3:DATA][d2:DATA][d1:DATA] =
(
       out[d1] -> PIPE_BEHAVIOR[d3][d2]
).

property FILTER_BUFFER_1_1(S=0,E=1) = FILTER_BUFFER_1_1[S],
FILTER_BUFFER_1_1[bufferCount:S..E] =
(
       when (bufferCount == S) pipeIn.out[DATA]
        -> FILTER_BUFFER_1_1[bufferCount + 1]
       | when (bufferCount > S && bufferCount < E) pipeIn.out[DATA]
        -> FILTER_BUFFER_1_1[bufferCount + 1]
       | when (bufferCount > S && bufferCount < E) pipeOut.in[DATA]
        -> FILTER_BUFFER_1_1[bufferCount - 1]
       | when (bufferCount == E) pipeOut.in[DATA]
        -> FILTER_BUFFER_1_1[bufferCount - 1]
).

property FILTER_1_1 =
(
       pipeIn.out[DATA] -> pipeOut.in[DATA] -> FILTER_1_1
).

property FILTER_1_N =
(
       pipeIn.out[DATA] -> pipeOut.in[DATA] -> FILTER_1_N_N
),
FILTER_1_N_N =
(
       pipeOut.in[DATA] -> FILTER_1_N_N
```

```
        | pipeIn.out[DATA] -> pipeOut.in[DATA] -> FILTER_1_N_N
).

property SINK_BEHAVIOR =
(
        pipeIn.out[DATA] -> SINK_BEHAVIOR
).

property PRODUCER_BEHAVIOR =
(
        pipeOut.in[DATA] -> PRODUCER_BEHAVIOR
).
```

8.7 Montecarlo_ClientServer_Signal.lts

Introducción

En este ejemplo se utilizan cuatro procesos principales correspondientes a la base de datos del proyecto, a las etapas de simulación y a los conectores de los estilos *Signal* y *Client-Server*. La base de datos del proyecto (servidor del conector *Client-Server*) se corresponde con el proceso `PROJECT_DATABASE` que resulta de la composición de los procesos `CS_SERVER_THREAD`, `TASK_DATA_STORE` y `CS_WRITE_LOCK`. El primero representa el comportamiento de los threads de este servidor, el segundo almacena los datos del proyecto y el tercero sincroniza las escrituras a estos datos. Cada una de las etapas de simulación se encuentra definida por el proceso `PROJECT_WORK_UNIT`, que se compone de los procesos `UNIT_RW`, `UNIT_PERMISSION_COUNT` y `CLIENT_RW`. El primero define las acciones que debe efectuar el componente: leer o escribir de/en la base de datos y notificar al siguiente componente en la cadena mediante una señal. El segundo determina los límites inferiores y superiores de los elementos de la base de datos sobre los que se puede operar sin "pisar" datos de otros componentes. El tercero define el comportamiento necesario para comunicarse con la base de datos del proyecto. El comportamiento del conector del estilo *Client-Server* es equivalente al definido y detallado en el listado `NewsService_ClientServer.lts`. El comportamiento de los conectores *Signal* es trivial y consiste en la comunicación de una única señal.

Código fuente

```
// ------------------------------------------------
// MONTECARLO SIMULATION EXAMPLE USING CS & SIGNALS
// ------------------------------------------------

const TASK_RISK_MIN = 1
const TASK_RISK_MAX = 2
range TASK_RISK = TASK_RISK_MIN..TASK_RISK_MAX

const SLOT_MIN = 0
const SLOT_MAX = 2
const SLOT_COUNT = (SLOT_MAX - SLOT_MIN + 1)
range SLOTS_COUNT = 0..SLOT_COUNT
range SLOTS = SLOT_MIN..SLOT_MAX

const INITIAL_DATA_VALUE = TASK_RISK_MIN

const RW_MIN = 1
const RW_MAX = 4
range RWS = RW_MIN..RW_MAX

const READ_PROJECT = 0
const WRITE_SUM = 2
const WRITE_PROJECT = 4

const CALL_READ = TASK_RISK_MIN - 1
range CALL_WRITE = TASK_RISK_MIN..TASK_RISK_MAX

const RESPONSE_OK = TASK_RISK_MIN - 1
const SERVER_RESPONSE_OK = TASK_RISK_MIN - 1
range SERVER_RESPONSES_DATA = TASK_RISK_MIN..TASK_RISK_MAX

const CLIENT_MIN = RW_MIN
const CLIENT_MAX = RW_MAX
range CLIENT_CALLS_WITH_SLOT_ARG = (TASK_RISK_MIN - 1)..TASK_RISK_MAX
const CLIENT_CALLS_WITH_SLOT_ARG_COUNT = TASK_RISK_MAX - (TASK_RISK_MIN -1) + 1
```

```
const SERVER_THREAD_MIN = 1
const SERVER_THREAD_MAX = 2

// PIPE + FILTER SPECIFICATION CONSTS
// ----------------------------------
const DATA_MIN = TASK_RISK_MIN
const DATA_MAX = TASK_RISK_MAX
range DATA = DATA_MIN..DATA_MAX

// CS SPECIFICATION CONSTS
// -----------------------
range CLIENTS = CLIENT_MIN..CLIENT_MAX
range CLIENT_CALLS = 0..(CLIENT_CALLS_WITH_SLOT_ARG_COUNT * SLOT_COUNT - 1)
range SERVER_RESPONSES = (TASK_RISK_MIN - 1)..TASK_RISK_MAX

const SERVER_THREAD_BUSY = SERVER_THREAD_MIN - 1
range SERVER_THREADS = SERVER_THREAD_MIN..SERVER_THREAD_MAX
range SERVER_THREADS_AND_BUSY = SERVER_THREAD_BUSY..SERVER_THREAD_MAX

// REFINED SYSTEM PROPERTIES
// -------------------------
property SIMULATION_PHASE_WORK =
(
        out[d:TASK_RISK] -> if(d == TASK_RISK_MAX) then
                                        (in[TASK_RISK_MIN] -> SIMULATION_PHASE_WORK)
                                else
                                        (in[d + 1]  -> SIMULATION_PHASE_WORK)
).

PUSH_PROJECT_TO_PIPELINE = (readTask[d:TASK_RISK] -> pipe[1].in[d] ->
PUSH_PROJECT_TO_PIPELINE).

PULL_PROJECT_FROM_PIPELINE = (pipe[3].out[d:TASK_RISK] -> writeTaskRisk[d] ->
PULL_PROJECT_FROM_PIPELINE).

// SYSTEM PROPERTIES
// -----------------
property RESULTS_PREVIEW_UPDATE =
(
        pdb[SERVER_THREADS].clientCall[CLIENTS][s:SLOTS][c:CALL_WRITE] ->
        updateResultPreviewData[s][c] ->
        RESULTS_PREVIEW_UPDATE
).

// SIGNAL CONX
// -----------
SIGNAL_CONX = (signal -> SIGNAL_CONX).

// CS CONNECTOR
// ------------
CS_CONX_ST_STATUS(T=0) = CS_CONX_ST_STATUS_FREE,
CS_CONX_ST_STATUS_FREE =
(
        pdb[T].clientCall[c:CLIENTS][SLOTS][CLIENT_CALLS_WITH_SLOT_ARG] ->
CS_CONX_ST_STATUS_BUSY[c]
),
CS_CONX_ST_STATUS_BUSY[c:CLIENTS] =
(
        pdb[SERVER_THREAD_BUSY].clientCall[CLIENTS][SLOTS][CLIENT_CALLS_WITH_SLOT_ARG]
         -> CS_CONX_ST_STATUS_BUSY[c]
        | pdb[T].clientResponse[c][SERVER_RESPONSES] -> CS_CONX_ST_STATUS_FREE
).

CS_CONX_CLIENT(C=0) =
(
        pwu[C].call[s:SLOTS][call:CLIENT_CALLS_WITH_SLOT_ARG] ->
        (
                pdb[st:SERVER_THREADS].clientCall[C][s][call]
                 -> pdb[st].clientResponse[C][r:SERVER_RESPONSES]
                 -> pwu[C].response[r]
```

```
                -> CS_CONX_CLIENT
               | pdb[SERVER_THREAD_BUSY].clientCall[C][s][call] -> CS_CONX_CLIENT
        )
).

|| CS_CONX =
(
        forall[t:SERVER_THREADS] CS_CONX_ST_STATUS(t)
        ||  forall[c:CLIENTS] CS_CONX_CLIENT(c)
)
/ {pwu[c:CLIENTS].dos /
pdb[SERVER_THREAD_BUSY].clientCall[c][SLOTS][CLIENT_CALLS_WITH_SLOT_ARG]}.

// PROJECT WORK UNIT
// ------------------
CLIENT_RW(BEHAVIOR = READ_PROJECT) =
(
        when (BEHAVIOR != READ_PROJECT) call[SLOTS][CALL_READ] ->
        (
                dos -> CLIENT_RW
                | response[SERVER_RESPONSES_DATA] -> CLIENT_RW
        )

        | when (BEHAVIOR != WRITE_PROJECT) call[SLOTS][CALL_WRITE] ->
        (
                dos -> CLIENT_RW
                | response[SERVER_RESPONSE_OK] -> CLIENT_RW
        )
) + {call[SLOTS][CALL_READ],call[SLOTS][CALL_WRITE]}.

UNIT_RW(U=RW_MIN,P=0,BEHAVIOR = READ_PROJECT) = UNIT_RW_READ[SLOT_MIN],
UNIT_RW_READ[nextSlot:SLOTS] =
(
        when(BEHAVIOR != READ_PROJECT) pwu[U].call[nextSlot][CALL_READ] ->
        (
                pwu[U].dos -> UNIT_RW_READ[nextSlot]
                | pwu[U].response[d:SERVER_RESPONSES_DATA] -> UNIT_RW_WRITE[nextSlot][d]

        )
        | when (BEHAVIOR == READ_PROJECT) readTask[d:TASK_RISK] ->
UNIT_RW_WRITE[nextSlot][d]
),
UNIT_RW_WRITE[nextSlot:SLOTS][d:TASK_RISK] =
(
        when (BEHAVIOR != READ_PROJECT) pwu[U].call[nextSlot][(TASK_RISK_MAX - d + 1)]
         -> UNIT_RW_WRITE_RESPONSE[nextSlot][d]
        | when (BEHAVIOR == READ_PROJECT) pwu[U].call[nextSlot][d]
         -> UNIT_RW_WRITE_RESPONSE[nextSlot][d]
        | when (BEHAVIOR == WRITE_PROJECT) writeTaskRisk[d] -> s[U].signal
         -> UNIT_RW_READ[(nextSlot + 1) % SLOT_COUNT]
),
UNIT_RW_WRITE_RESPONSE[nextSlot:SLOTS][d:TASK_RISK] =
(
        pwu[U].dos -> UNIT_RW_WRITE[nextSlot][d]
        | pwu[U].response[SERVER_RESPONSE_OK] -> s[U].signal -> UNIT_RW_READ[(nextSlot + 1)
% SLOT_COUNT]
).

UNIT_PERMISSION_COUNT(U=RW_MIN,P=0,IN_SIGNAL=RW_MAX,BEHAVIOR = READ_PROJECT) =
UNIT_PERMISSION_COUNT[P],
UNIT_PERMISSION_COUNT[permissionCount:SLOTS_COUNT] =
(
        when (permissionCount > 0 && BEHAVIOR == READ_PROJECT)
                readTask[d:TASK_RISK] -> UNIT_PERMISSION_COUNT[permissionCount]
        | when (permissionCount > 0 && BEHAVIOR != READ_PROJECT)
                pwu[U].call[SLOTS][CALL_READ] -> UNIT_PERMISSION_COUNT[permissionCount]
        | when (permissionCount < SLOT_COUNT)
                s[IN_SIGNAL].signal -> UNIT_PERMISSION_COUNT[permissionCount + 1]
        | when (permissionCount > 0)
                s[U].signal -> UNIT_PERMISSION_COUNT[permissionCount - 1]
).
```

```
|| PROJECT_WORK_UNIT(U=RW_MIN,P=0,IN_SIGNAL=RW_MAX,BEHAVIOR=WRITE_PROJECT) =
(
        UNIT_RW(U,P,BEHAVIOR)
        || UNIT_PERMISSION_COUNT(U,P,IN_SIGNAL,BEHAVIOR)
        || pwu[U]:CLIENT_RW(BEHAVIOR)
).

// PROJECT DATABASE
// ----------------
CS_SERVER_THREAD(T=SERVER_THREAD_MIN) =
(
        pdb[T].clientCall[c:CLIENTS][s:SLOTS][CALL_READ]
         -> pdb[T].clientResponse[c][s][SERVER_RESPONSES_DATA]
         -> CS_SERVER_THREAD
        | pdb[T].clientCall[c:CLIENTS][s:SLOTS][w:CALL_WRITE]
         -> pdb[T].clientWriteResponse[c][s][SERVER_RESPONSE_OK]
         -> CS_SERVER_THREAD
).

CS_WRITE_LOCK =
(
        pdb[t:SERVER_THREADS].clientCall[c:CLIENTS][s:SLOTS][w:CALL_WRITE]
         -> updateResultPreviewData[s][w]
         -> pdb[t].clientWriteResponse[c][s][SERVER_RESPONSE_OK]
         -> CS_WRITE_LOCK
).

TASK_DATA_STORE(S=SLOT_MIN) = TASK_DATA_STORE[INITIAL_DATA_VALUE],
TASK_DATA_STORE[d:TASK_RISK] =
(
        pdb[SERVER_THREADS].clientCall[CLIENTS][S][w:CALL_WRITE] -> TASK_DATA_STORE[w]
        | pdb[SERVER_THREADS].clientResponse[CLIENTS][S][d] -> TASK_DATA_STORE[d]
).
|| PROJECT_DATABASE_A =
(
        forall[t:SERVER_THREADS] CS_SERVER_THREAD(t)
        || forall[s:SLOTS] TASK_DATA_STORE(s)
        || CS_WRITE_LOCK
).

|| PROJECT_DATABASE =
(
        PROJECT_DATABASE_A
)
/ {
        pdb[t:SERVER_THREADS].clientResponse[c:CLIENTS][s:SERVER_RESPONSES]
         / pdb[t].clientWriteResponse[c][SLOTS][s],
        pdb[t:SERVER_THREADS].clientResponse[c:CLIENTS][s:SERVER_RESPONSES]
         / pdb[t].clientResponse[c][SLOTS][s]
}.

// ARCHITECTURE
// ------------
|| ARQREF =
(
        PROJECT_DATABASE
        || PROJECT_WORK_UNIT(1,SLOT_COUNT,4,READ_PROJECT)
        || PROJECT_WORK_UNIT(2,0,1,WRITE_SUM)
        || PROJECT_WORK_UNIT(3,0,2,WRITE_SUM)
        || PROJECT_WORK_UNIT(4,0,3,WRITE_PROJECT)
        || CS_CONX
        || forall[rw:RWS] s[rw]:SIGNAL_CONX
).

|| CHECKED_ARQREF =
(
        ARQREF
        || RESULTS_PREVIEW_UPDATE
).
```

```
// ARCHITECTURE IS CS
// ------------------
TRANSLATE_CLIENT =
(
        pwu[c:CLIENTS].call[s:SLOTS][call:CLIENT_CALLS_WITH_SLOT_ARG]
         -> client[c].call[call * (SLOT_COUNT) + s]
         -> TRANSLATE_CLIENT
        | pwu[c:CLIENTS].dos -> client[c].dos -> TRANSLATE_CLIENT
        | pwu[c:CLIENTS].response[s:SERVER_RESPONSES] -> client[c].response[s]
         -> TRANSLATE_CLIENT
).

TRANSLATE_ASSOCIATE =
(
        pdb[t:SERVER_THREADS].clientCall[c:CLIENTS][s:SLOTS][call:CLIENT_CALLS_WITH_SLOT_ARG
]
         -> associate[t][c] -> st[t].clientCall[call * (SLOT_COUNT) + s]
         -> TRANSLATE_ASSOCIATE
).

TRANSLATE_DISASSOCIATE =
(
        pdb[t:SERVER_THREADS].clientResponse[c:CLIENTS][response:SERVER_RESPONSES]
         -> st[t].clientResponse[response]
         -> disassociate[t][c]
         -> TRANSLATE_DISASSOCIATE
).

|| ARQ_IS_CS =
(
        ARQREF
        || TRANSLATE_CLIENT
        || TRANSLATE_ASSOCIATE
        || TRANSLATE_DISASSOCIATE
        || forall[c:CLIENTS] client[c]:CLIENT_BEHAVIOR
        || forall[t:SERVER_THREADS] SERVER_THREAD_BEHAVIOR(t)
        || forall[t:SERVER_THREADS] SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR(t)
        || forall[c:CLIENTS] ASSOCIATION_KEEPS_CLIENT_CALL(c)
        || forall[c:CLIENTS] ASSOCIATION_KEEPS_SERVER_RESPONSE(c)
)
<< {
        client[CLIENTS].call[CLIENT_CALLS],
        client[CLIENTS].dos,
        client[CLIENTS].response[SERVER_RESPONSES],
        associate[t:SERVER_THREADS][c:CLIENTS],
        disassociate[t:SERVER_THREADS][c:CLIENTS],
        st[SERVER_THREADS].clientCall[CLIENT_CALLS],
        st[SERVER_THREADS].clientResponse[SERVER_RESPONSES]
}.

// -------------------------
// ARCHITECTURE IS REFINEMENT
// -------------------------

// TRANSLATION
// -----------
const PIPE_MIN = 1
const PIPE_MAX = 3
const PIPE_COUNT = PIPE_MAX - PIPE_MIN + 1
range PIPES = PIPE_MIN..PIPE_MAX


TRANSLATE_IN(U=RW_MIN,PIPE=0) =
(
        pwu[U].call[SLOTS][w:CALL_WRITE] ->
        (
                pwu[U].call[SLOTS][CALL_WRITE] -> TRANSLATE_IN
                | pwu[U].dos -> TRANSLATE_IN
                | pwu[U].response[SERVER_RESPONSE_OK] -> pipe[PIPE].in[w] -> TRANSLATE_IN
        )
        | pwu[U].dos -> TRANSLATE_IN
```

```
        | pwu[U].response[SERVER_RESPONSE_OK] -> TRANSLATE_IN
).
TRANSLATE_OUT(U=RW_MIN,PIPE=0) =
(
        pwu[U].call[SLOTS][CALL_READ] ->
        (
                pwu[U].call[SLOTS][CALL_READ] -> TRANSLATE_OUT
                | pwu[U].dos -> TRANSLATE_OUT
                | pwu[U].response[d:SERVER_RESPONSES_DATA] -> pipe[PIPE].out[d] ->
TRANSLATE_OUT
        )
        | pwu[U].dos -> TRANSLATE_OUT
        | pwu[U].response[SERVER_RESPONSES_DATA] -> TRANSLATE_OUT
).

//progress PIPELINE_IS_WORKING = {pipe[PIPES].in[DATA], pipe[PIPES].out[DATA]}

|| ARQ_AS_PIPELINE =
(
        ARQREF

        || TRANSLATE_IN(1,1)
        || TRANSLATE_OUT(2,1)
        || TRANSLATE_IN(2,2)
        || TRANSLATE_OUT(3,2)
        || TRANSLATE_IN(3,3)
        || TRANSLATE_OUT(4,3)
) << {
        pipe[PIPES].in[DATA],
        pipe[PIPES].out[DATA]
}.

|| ARQ_IS_REFINEMENT =
(
        ARQ_AS_PIPELINE
        || pipe[1]::PIPE_BEHAVIOR
        || pipe[2]::PIPE_BEHAVIOR
        || pipe[3]::PIPE_BEHAVIOR
        || PRODUCER_BEHAVIOR
                / { pipe[1].in[d:DATA] / pipeOut.in[d]}
        || SINK_BEHAVIOR
                 / { pipe[3].out[d:DATA] / pipeIn.out[d]}
        || FILTER_1_1
                / { pipe[1].out[d:DATA] / pipeIn.out[d], pipe[2].in[d:DATA] / pipeOut.in[d]}
        || FILTER_1_1
                / { pipe[2].out[d:DATA] / pipeIn.out[d], pipe[3].in[d:DATA] / pipeOut.in[d]}
        || FILTER_BUFFER_1_1(0,6)
                / { pipe[1].out[d:DATA] / pipeIn.out[d], pipe[3].in[d:DATA] / pipeOut.in[d]}
        || SIMULATION_PHASE_WORK
                / { pipe[1].out[d:TASK_RISK] / out[d], pipe[2].in[d:TASK_RISK] / in[d]}
        || SIMULATION_PHASE_WORK
                / { pipe[2].out[d:TASK_RISK] / out[d], pipe[3].in[d:TASK_RISK] / in[d]}
        || PUSH_PROJECT_TO_PIPELINE
        || PULL_PROJECT_FROM_PIPELINE
).

// ---------------------------------
// CLIENT-SERVER STYLE SPECIFICATION
// ---------------------------------

#INCLUDE “ClientServer.lts”

// ---------------------------------
// PIPE + FILTER STYLE SPECIFICATION
// ---------------------------------

#INCLUDE “PipeFilter.lts”
```

8.8 Turns_SharedVar.lts

Introducción

Este ejemplo se compone de tres procesos principales correspondientes a los jugadores (lecto/escritores de la variable) a la variable propiamente dicha y al conector del estilo utilizado. Cada uno de los jugadores se encuentra representado por un proceso `RW`. Este proceso consiste en la lectura continua de la variable (`read[d:DATA]`) a la espera de encontrar el turno asignado al mismo (`DATA_MY_TURN`), para luego escribir en la variable el turno del próximo jugador (`DATA_NEXT_TURN`). La variable se encuentra modelada por el proceso homónimo y su comportamiento es el esperable. El comportamiento del conector del estilo se encuentra representado por el proceso `VAR_CONX` y consiste en la sincronización del acceso (lectura o escritura) a la variable.

Código fuente

```
// -----------------------------------------
// | (!) TURNS EXAMPLE USING SHARED VARIABLE |
// -----------------------------------------

// CONSTS FOR SHARED VARIABLE SPECS
// --------------------------------
const RWS_MIN = 0
const RWS_MAX = 2
range RWS = RWS_MIN..RWS_MAX
const DATA_MIN = RWS_MIN
const DATA_MAX = RWS_MAX
const INITIAL_DATA_VALUE = DATA_MIN
range DATA = DATA_MIN..DATA_MAX

// CANONNICAL IMPLEMENTATION OF SHARED VARIABLE
// --------------------------------------------

RW(DATA_MY_TURN=DATA_MIN, DATA_NEXT_TURN=DATA_MAX) =
(
        read[d:DATA]
         -> if (d == DATA_MY_TURN) then (turn -> write[DATA_NEXT_TURN] -> RW) else RW
)
+ {read[DATA], write[DATA]}.

VARIABLE = VARIABLE[INITIAL_DATA_VALUE],
VARIABLE[d:DATA] =
        (varRead[d] -> VARIABLE[d] | varWrite[dataWrite:DATA] -> VARIABLE[dataWrite]).

VAR_CONX =
(
        rw[RWS].write[w:DATA] -> VAR_CONX
        | rw[RWS].read[d:DATA] -> VAR_CONX
).

// SYSTEM PROPS
// ------------
property TURNS = TURNS[RWS_MIN],
TURNS[t:RWS] = (rw[t].turn -> TURNS[(t + 1) % (RWS_MAX + 1)]).

progress TURN_PROGRESS = {rw[RWS].turn}

// ARCHITECTURE
// ------------
|| ARQ =
(
        forall[rw:RWS] rw[rw]:RW(rw, (rw + 1) % (RWS_MAX + 1))
        || VAR_CONX
```

```
        || VARIABLE
         / { rw[RWS].write[w:DATA] / varWrite[w], rw[RWS].read[d:DATA] / varRead[d]}
).

|| CHECKED_ARQ =
(
        ARQ
        || TURNS
).

|| ARQ_IS_SHARED_VARIABLE =
(
        ARQ
        || forall[rw:RWS] RW_BEHAVIOR(rw)
        || RWS_VARIABLE_BEHAVIOR
).

// ---------------------------------
// SHARED VARIABLE STYLE SPECIFICATION
// ---------------------------------

#INCLUDE “SharedVariable.lts”
```

8.9 SharedVariable.lts

Introducción

La descripción de cada uno de los procesos/propiedades detallados en este listado puede encontrarse en el catálogo de estilos del Apéndice A.

Código fuente

```
// ---------------------------------
// SHARED VARIABLE STYLE SPECIFICATION
// ---------------------------------

// NOTES: RW stands for READER/WRITER

/*
 ROLES:
       1 - READER/WRITER (1..*)
       2 - VARIABLE VAULT (1)

 CONNECTORS:
       1 - CS SHARE VARIABLE (1)
              -> Role RW (1..*) - READER/WRITER
              -> Role Variable (1) - VARIABLE VAULT

 CONFIGURATION:
       - Restrictions: none beyond those imposed by the connectors
*/

property RWS_VARIABLE_BEHAVIOR = RWS_VARIABLE_BEHAVIOR[INITIAL_DATA_VALUE],
RWS_VARIABLE_BEHAVIOR[d:DATA] =
(
       rw[RWS].read[d] -> RWS_VARIABLE_BEHAVIOR[d]
       | rw[RWS].write[w:DATA] -> RWS_VARIABLE_BEHAVIOR[w]
).

property RW_BEHAVIOR(RW=1) = (rw[RW].read[d:DATA] -> RW_BEHAVIOR | rw[RW].write[d:DATA] ->
RW_BEHAVIOR).
```

8.10 Turns_ClientServer.lts

Introducción

Este ejemplo se compone de tres procesos principales correspondientes a los jugadores (lecto/escritores de la variable) al servidor que almacena la variable y al conector del estilo utilizado. Cada uno de los jugadores se encuentra representado por un proceso `CLIENT_RW`. Este proceso consiste en la lectura continua de la variable mediante llamadas `call[CALL_READ]` al servidor, a la espera de encontrar el turno asignado al mismo (`MY_TURN`). Luego, se escribe un nuevo valor para la variable mediante un llamado `call[(MY_TURN + 1) % (RWS_MAX + 1)]` al servidor. El servidor se encuentra modelado mediante el proceso `SERVER` que resulta de la composición de los procesos `SERVER_THREAD` y `SERVER_VARIABLE`. El primero modela la recepción y el procesamiento de las llamadas de los clientes. El segundo mantiene el valor de la variable y sincroniza el acceso a la misma por parte de los threads. También se provee un comportamiento alternativo para el servidor mediante el proceso `FAULT_SERVER`, por el cual se responde erróneamente a las llamadas de los clientes. El comportamiento del conector es equivalente al definido y detallado en el listado `NewsService_ClientServer.lts`.

Código fuente

```
// -----------------------------
// | (!) TURNS EXAMPLE USING CS |
// -----------------------------

// CONSTS FOR SHARED VARIABLE SPECS
// --------------------------------
const RWS_MIN = 0
const RWS_MAX = 2
range RWS = RWS_MIN..RWS_MAX

const DATA_MIN = RWS_MIN
const DATA_MAX = RWS_MAX
const INITIAL_DATA_VALUE = DATA_MIN
range DATA = DATA_MIN..DATA_MAX

// Refinamiento utilizando CS
// --------------------------

const CALL_READ = DATA_MIN - 1
range CALL_WRITE = DATA_MIN..DATA_MAX

const RESPONSE_OK = DATA_MIN - 1

const CLIENT_MIN = RWS_MIN
const CLIENT_MAX = RWS_MAX
range CLIENTS = CLIENT_MIN..CLIENT_MAX

const SERVER_THREAD_MIN = 1
const SERVER_THREAD_MAX = 2
const SERVER_THREAD_BUSY = SERVER_THREAD_MIN - 1
range SERVER_THREADS = SERVER_THREAD_MIN..SERVER_THREAD_MAX

range CLIENT_CALLS = (DATA_MIN - 1)..DATA_MAX

const SERVER_RESPONSE_OK = DATA_MIN - 1
range SERVER_RESPONSES_DATA = DATA_MIN..DATA_MAX
range SERVER_RESPONSES = (DATA_MIN - 1)..DATA_MAX

// CS CONNECTOR
```

```
// ------------

CS_CONX_ST_STATUS(T=0) = CS_CONX_ST_STATUS_FREE,
CS_CONX_ST_STATUS_FREE =
(
        st[T].call[c:CLIENTS][CLIENT_CALLS] -> CS_CONX_ST_STATUS_BUSY[c]
),
CS_CONX_ST_STATUS_BUSY[c:CLIENTS] =
(
        st[SERVER_THREAD_BUSY].call[CLIENTS][CLIENT_CALLS] -> CS_CONX_ST_STATUS_BUSY[c]
        | st[T].response[c][SERVER_RESPONSES] -> CS_CONX_ST_STATUS_FREE
).

CS_CONX_CLIENT(C=0) =
(
        client[C].call[call:CLIENT_CALLS] ->
        (
                st[s:SERVER_THREADS].call[C][call]
                 -> st[s].response[C][r:SERVER_RESPONSES]
                 -> client[C].response[r]
                 -> CS_CONX_CLIENT
                | st[SERVER_THREAD_BUSY].call[C][call] -> CS_CONX_CLIENT
        )
).

|| CS_CONX =
(
        forall[t:SERVER_THREADS] CS_CONX_ST_STATUS(t)
        || forall[c:CLIENTS] CS_CONX_CLIENT(c)
)
/ {client[c:CLIENTS].dos / st[SERVER_THREAD_BUSY].call[c][CLIENT_CALLS]}.

// CS CLIENT (RW)
// --------------

CLIENT_RW(MY_TURN=DATA_MIN) = CLIENT_RW_POLL_TURN,
CLIENT_RW_POLL_TURN =
(
        call[CALL_READ] ->
        (
                dos -> CLIENT_RW
                | response[r:DATA_MIN..DATA_MAX] -> if (r == MY_TURN) then
                                                        (turn -> CLIENT_WRITE_MY_TURN)
                                                    else
                                                        CLIENT_RW_POLL_TURN
        )
),
CLIENT_WRITE_MY_TURN =
(
        call[(MY_TURN + 1) % (RWS_MAX + 1)] ->
        (
                dos -> CLIENT_WRITE_MY_TURN
                | response[SERVER_RESPONSE_OK] -> CLIENT_RW
        )
) + {call[CLIENT_CALLS]}.

// CS SERVER (Variable Vault)
// --------------------------

SERVER_THREAD(T=SERVER_THREAD_MIN) =
(
        st[T].call[c:CLIENTS][CALL_READ] -> st[T].response[c][SERVER_RESPONSES_DATA]
         -> SERVER_THREAD
        | st[T].call[c:CLIENTS][w:CALL_WRITE] -> st[T].response[c][SERVER_RESPONSE_OK]
         -> SERVER_THREAD
) + {st[T].response[CLIENTS][SERVER_RESPONSES]}.

SERVER_VARIABLE = SERVER_VARIABLE[INITIAL_DATA_VALUE],
SERVER_VARIABLE[d:DATA] =
(
        st[SERVER_THREADS].call[CLIENTS][w:CALL_WRITE] -> SERVER_VARIABLE[w]
```

```
        | st[SERVER_THREADS].call[CLIENTS][CALL_READ] -> SERVER_VARIABLE[d]
        | st[SERVER_THREADS].response[CLIENTS][d] -> SERVER_VARIABLE[d]
        | st[SERVER_THREADS].response[CLIENTS][SERVER_RESPONSE_OK] -> SERVER_VARIABLE[d]
).

|| SERVER =
(
        forall[t:SERVER_THREADS] SERVER_THREAD(t)
        || SERVER_VARIABLE
).

// SYSTEM PROPS
// ------------
property TURNS = TURNS[RWS_MIN],
TURNS[t:RWS] = (rw[t].turn -> TURNS[(t + 1) % (RWS_MAX + 1)]).

progress TURN_PROGRESS = {rw[RWS].turn}

// SYSTEM
// ------

|| ARQ =
(
        forall[c:CLIENTS] client[c]:CLIENT_RW(c) / {rw[c].turn / client[c].turn}
        || CS_CONX
        || SERVER
).

TRANSLATE_RW =
(
        st[SERVER_THREADS].call[c:CLIENTS][w:CALL_WRITE] -> rw[c].write[w] -> TRANSLATE_RW
        | st[SERVER_THREADS].response[c:CLIENTS][d:SERVER_RESPONSES_DATA] -> rw[c].read[d]
         -> TRANSLATE_RW
).

CANNONICAL_CS_TRANSLATION_ASSOCIATE =
(
        st[t:SERVER_THREADS].call[c:CLIENTS][call:CLIENT_CALLS] -> associate[t][c]
         -> st[t].clientCall[call] -> CANNONICAL_CS_TRANSLATION_ASSOCIATE
).
CANNONICAL_CS_TRANSLATION_DISASSOCIATE =
(
        st[t:SERVER_THREADS].response[c:CLIENTS][response:SERVER_RESPONSES]
         -> st[t].clientResponse[response]
         -> disassociate[t][c]
         -> CANNONICAL_CS_TRANSLATION_DISASSOCIATE
).

|| ARQ_IS_CS =
(
        ARQ
        || CANNONICAL_CS_TRANSLATION_ASSOCIATE
        || CANNONICAL_CS_TRANSLATION_DISASSOCIATE

        || forall[c:CLIENTS] client[c]:CLIENT_BEHAVIOR
        || forall[t:SERVER_THREADS] SERVER_THREAD_BEHAVIOR(t)
        || forall[t:SERVER_THREADS] SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR(t)
        || forall[c:CLIENTS] ASSOCIATION_KEEPS_CLIENT_CALL(c)
        || forall[c:CLIENTS] ASSOCIATION_KEEPS_SERVER_RESPONSE(c)
) << {
        associate[t:SERVER_THREADS][c:CLIENTS],
        disassociate[t:SERVER_THREADS][c:CLIENTS],
        st[SERVER_THREADS].clientCall[CLIENT_CALLS],
        st[SERVER_THREADS].clientResponse[SERVER_RESPONSES]
}.

// CHECKED SYSTEM
// --------------

|| CHECKED_ARQ =
(
```

```
        ARQ
        || TURNS
        || TRANSLATE_RW
        || RWS_VARIABLE_BEHAVIOR
        || forall[rw:RWS] RW_BEHAVIOR(rw)
)
<< {rw[CLIENTS].read[DATA], rw[CLIENTS].write[DATA]}.

// FAULTY
// -------

FAULT_SERVER_THREAD(T=SERVER_THREAD_MIN) =
(
        st[T].call[c:CLIENTS][CALL_READ]
         -> st[T].response[c][SERVER_RESPONSES_DATA]
         -> FAULT_SERVER_THREAD
        | st[T].call[c:CLIENTS][w:CALL_WRITE]
         -> st[T].response[c][SERVER_RESPONSE_OK]
         -> FAULT_SERVER_THREAD
).

FAULT_SERVER_VARIABLE = FAULT_SERVER_VARIABLE[INITIAL_DATA_VALUE],
FAULT_SERVER_VARIABLE[d:DATA] =
(
      st[SERVER_THREADS].call[CLIENTS][w:CALL_WRITE] -> FAULT_SERVER_VARIABLE[w]
      | st[SERVER_THREADS].call[CLIENTS][CALL_READ] -> FAULT_SERVER_VARIABLE[d]
      | st[SERVER_THREADS].response[CLIENTS][DATA_MIN] -> FAULT_SERVER_VARIABLE[d]
      | st[SERVER_THREADS].response[CLIENTS][SERVER_RESPONSE_OK] -> FAULT_SERVER_VARIABLE[d]
)
+ {st[SERVER_THREADS].response[CLIENTS][SERVER_RESPONSES]}.

|| FAULT_SERVER =
(
        forall[t:SERVER_THREADS] FAULT_SERVER_THREAD(t)
        || FAULT_SERVER_VARIABLE
).

|| FAULT_ARQ =
(
        forall[c:CLIENTS] client[c]:CLIENT_RW(c) / {rw[c].turn / client[c].turn}
        || CS_CONX
        || FAULT_SERVER
).

|| FAULT_ARQ_IS_CS =
(
        FAULT_ARQ
        || CANNONICAL_CS_TRANSLATION_ASSOCIATE
        || CANNONICAL_CS_TRANSLATION_DISASSOCIATE

        || forall[c:CLIENTS] client[c]:CLIENT_BEHAVIOR
        || forall[t:SERVER_THREADS] SERVER_THREAD_BEHAVIOR(t)
        || forall[t:SERVER_THREADS] SERVER_THREAD_CLIENT_ASSOCIATION_BEHAVIOR(t)
        || forall[c:CLIENTS] ASSOCIATION_KEEPS_CLIENT_CALL(c)
        || forall[c:CLIENTS] ASSOCIATION_KEEPS_SERVER_RESPONSE(c)
) << {
        associate[t:SERVER_THREADS][c:CLIENTS],
        disassociate[t:SERVER_THREADS][c:CLIENTS],
        st[SERVER_THREADS].clientCall[CLIENT_CALLS],
        st[SERVER_THREADS].clientResponse[SERVER_RESPONSES]
}.

// CHECKED SYSTEM
// --------------

|| FAULT_CHECKED_ARQ =
(
        FAULT_ARQ
        || TRANSLATE_RW
        || RWS_VARIABLE_BEHAVIOR
        || forall[rw:RWS] RW_BEHAVIOR(rw)
```

```
)
<< {rw[CLIENTS].read[DATA], rw[CLIENTS].write[DATA]}.

|| FAULT_CHECKED_ARQ_WITH_SYS_PROPS =
(
        FAULT_ARQ
        || TURNS
        || TRANSLATE_RW
        || RWS_VARIABLE_BEHAVIOR
        || forall[rw:RWS] RW_BEHAVIOR(rw)
)
<< {rw[CLIENTS].read[DATA], rw[CLIENTS].write[DATA]}.

// ------------------------------------
// SHARED VARIABLE STYLE SPECIFICATION
// ------------------------------------

#INCLUDE “SharedVariable.lts”

// ------------------------------------
// CLIENT-SERVER STYLE SPECIFICATION
// ------------------------------------

#INCLUDE “ClientServer.lts”
```

9. Bibliografía

[AAG93] Gregory Abowd, Robert Allen, David Garlan. Using style to understand descriptions of software architecture. In Proceedings of SIGSOFT'93: Foundations of Software Engineering, Software Engineering Notes 18(5), pages 9–20. ACMPress, December 1993.

[AAG95] Gregory Abowd, Robert Allen, David Garlan. Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering and Methodology, 4(4):319–64, October 1995.

[AG92] Robert Allen, David Garlan. A formal approach to software architectures. In Jan van Leeuwen, editor, Proceedings of IFIP'92, pages 134–41. Elsevier Science Publishers B.V., September 1992.

[Alex77] Christopher Alexander. The Timeless Way of Building. Oxford University Press, 1977

[Alex79] Christopher Alexander. A Pattern Language: Towns, Buildings, Construction. Oxford University Press 1979

[AM99] Marwan Abi-Antoun, Nenad Medvidovic. Enabling the Refinement of a Software Architecture into a Design. In Proceedings of the 2nd International Conference on The Unified Modeling Language (UML'99), pp 17-31, Fort Collins, CO, October 28-30, 1999

[Bach95] J. Bach. Enough About Process: What We Need are Heroes. IEEE Software, Volume 12, Issue 2, pp 96-98, 1995

[BCK03] Len Bass, Paul Clements, Rick Kazman. Software Architecture in Practice, 2nd Edition. Addison-Wesley Professional, 2003

[Beck99] Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 1999

[BRJ99] Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley Professional, 1999

[Brooks87] F. Brooks. No Silver Bullet, Essence and Accidents in Software Engineering, IEEE Computer, Abril 1987

[Car98] A. Carzaniga. Architectures for an Event Notification Service Scalable to Wide-area Networks. Ph.D. thesis, Politecnico di Milano, Milano, Italy. 1998

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford. Documenting Software Architectures: Views and Beyond. Addison-Wesley Professional, 2002

[Chrome] Google Chrome Web Site. http://www.google.com/googlebooks/chrome/

[CES86] E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In ACM Transactions on Programming Languages and Systems, 8(2):244--263, 1986

[CKK01] P. Clements, R. Kazman, M. Klein. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley Professional, 2001

[CPT99] C. Canal, E. Pimentel, J. M. Troya. Specification and Refinement of Dynamic Software Architectures. First Working IFIP Conference on Software Architecture. 1999

[DK76] Frank DeRemer, Hans Kron. Programming-in-the-large versus programming-in-the-small. IEEE transactions on software engineering, SE-2(2):80-86. Junio 1976

[FDR05] Formal Systems (Europe). Failures-Divergence Refinement, FDR2 User Manual. Formal Systems (Europe) Ltd, 2005

[Fowler03] Martin Fowler. Who needs an architect? IEEE Software. Volume 20. Issue 5. pp: 11-13. Septiembre 2003

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In Proceedings of the 17th International Conference on Software Engineering, Seattle,Washington, April 1995.

[Gar96] David Garlan. Style-based refinement for software architecture. In Second International Software Architecture Workshop (ISAW-2), pages 72–75, San Fransisco, October 1996. ACM SIGSOFT.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Design. Addison-Wesley, 1995

[GMW00] D. Garlan, R.T. Monroe, D, Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, G.T. Leavens, M. Sitaraman (eds), Cambridge University Press, 2000

[GS93] David Garlan , Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, Advances in Software Engineering and Knowledge Engineering, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[GS96] David Garlan, Mary Shaw. Software Architecture: Perspectives on an emerging discipline. Prentice Hall, 1996

[HNS99] Christine Hofmeister, Robert Norde, Dilip Soni. Applied Software Architecture. Addison-Wesley Professional, 1999

[Hoa85] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

[Hoh03] Luke Hohmann. Beyond Software Architecture: Creating and Sustaining Winning Solutions. Addison-Wesley Professional, 2003

[Holt01] R. Holt. Software Architecture as a Shared Mental Model ASERC Workhop on Software Architecture, University of Alberta, Agosto, 2001.

[Holte] Holte consulting Web Site. http://www.holteconsulting.no/

[Hype] Gartner Web Site, http://www.gartner.com/pages/story.php.id.8795.s.8.jsp

[JBR99] Ivar Jacobson, Grady Booch, James Rumbaugh. The Unified Software Development Process. Addison-Wesley Professional, 1999

[KBNB04] Rilla Khaled, Pippin Barr, James Noble, Robert Biddle. Extreme Programming System Metaphor: A Semiotic Approach. International Workshop on Organisational Semiotics (pp. 152 172). Setubal, Portugal: INSTICC Press, 2004

[KJ99] R. Kazman, S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. Automated Software Engineering, Apr. 1999.

[Kru95] P. Kruchten. The 4+1 View Model of architecture, IEEE Software, vol. 12, issue. 6 pp. 42 -50, 1995

[Liu00] C. Liu. Smalltalk, objects, and design. AuthorHouse, 2000

[McConnell04] Steve McConnell. Code Complete: A Practical Handbook of Software Construction, 2nd Edition. Microsoft Press, 2004

[MK99] Jeff Magee, Jeff Kramer. Concurrency: State Models and Java Programs. John Wiley & Sons Ltd., New York, 1999.

[MNS95] G. Murphy, D. Notkin, K. Sullivan, “Software Reflexion Models: Bridging the Gap between Source and High-Level Models”, Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, (Washington, D.C.), October 1995

[MQ94] Mark Moriconi, Xiaolei Qian. Correctness and composition of software architectures. In Proceedings of SIGSOFT ’94: 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 164–74, New Orleans,LA, December 1994.

[MQR95] M. Moriconi, X. Qian, R. Riemenschneider. Correct architecture refinement. IEEE Transactions on Software Engineering, 21(4):356–372, April 1995.

[Naur85] P. Naur. Programming as theory building, Microprocessing and Microprogramming, vol. 15, 253-261, 1985.

[Par72] David Parnas. On the Criteria to be Used in decomposing Systems into Modules, Communications of the ACM, vol. 15(2), 1972.

[Par79] David Parnas. Designing software for ease of extension and contraction. IEEE Transactions on software engineering, 5:128-138, Marzo 1979

[PMI04] Project Management Institute. A Guide to the Project Management Body of Knowledge (PMBOK® Guide) - 3rd Edition. Project Management Institute, 2004

[PP6] Primavera P6 Web Site. http://www.primavera.com/products/p6/index.asp

[PW92] D.E. Perry , A.L. Wolf, Foundations for the study of Software Architectures, ACM SIGSOFT, Software Engineering Notes, 17 (4), 1992, pp 40-52.

[Ros97] A. Roscoe. Theory and Practice of Concurrency. Prentice Hall, 1997

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering, 21(4):314–335, April 1995

[SEI] SEI Web Site, Software architecture for software-intensive systems - How Do You Define Software Architecture? http://www.sei.cmu.edu/architecture/definitions.html

[SG04] B. Schmerl, D. Garlan. Acmestudio: Supporting style-centered architecture development. In Proc. Of the 26th International Conference on Software Engineering (ICSE), 2004

[Str00] Bjarne Stroustrup. The C++ Programming Language: Special Edition , 3rd Edition Addison-Wesley Professional, 2000

[SW99] Judith Stafford, Alexander Wolf. Architecture-Based Software Engineering.University of colorado, Department of Computer Science Technical Report CU-CS-891-99. Noviembre 1999

[UBC07] Sebastian Uchitel, Greg Brunet, Marsha Chechik, *Behaviour Model Synthesis from Properties and Scenarios* 29th IEEE/ACM International Conference on Software Engineering (ICSE), Minneapolis, 2007

[YGS+04] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, Rick Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, 23-28 Mayo 2004