UNA ARQUITECTURA FLEXIBLE ORIENTADA HACIA EL AGENTE PARA ANÁLISIS DE INTERACCIÓN Y RETROALIMENTACIÓN

H. Ulrich Hoppe, Stefan Weinbrenner y Lars Bollen[9]

RESUMEN

Se describe un marco arquitectónico para la ingeniería de ambientes de aprendizaje distribuido, que permite acoplar agentes de soporte inteligente programados con un agente de lenguajes múltiples. El marco implementa una arquitectura de pizarrón con un servidor de Espacio *Tuple* central basado en Java y clientes que difieren en el hardware (PDAs, PCs con proyección) y posiblemente en los lenguajes de programación (C#, *Prolog*, Java). Basado en esto, es posible analizar y apoyar actividades de aprendizaje a partir de logaritmos de acción así como información del estado del sistema. Este marco fue utilizado para respaldar discusiones en grupo que incluyen reuniones de diseño con dispositivos móviles de entrada y un gran despliegue público. Se han implementado agentes analíticos en *Prolog.*

Palabras clave: análisis con colaboración, dispositivos móviles, espacios *tuple*, racionamiento del diseño, apoyo en discusiones.

ABSTRACT

An architectural framework for the engineering of distributed learning environments is described, which allows for plugging in intelligent support agents programmed in multiple languages agent. The framework implements a blackboard architecture with a Java-based central *Tuple* Space server and clients that differ in hardware (PDAs, PCs with projection) and possibly in programming languages (C#, *Prolog*, Java). On this basis, it is possible to analyze and flexibly support learning activities based on action logs as well as system state information. This framework has been used for supporting group discussions including design meetings with mobile input devices and a large public display. Analytic agents have been implemented in *Prolog*.

Key words: Collaboration analysis, mobile devices, *tuple* spaces, design rationale, discussion support.

Recibido: 18 de agosto de 2009
Aceptado: 28 de abril de 2010

9 Departamento de Informática y Ciencia Cognitiva Aplicada . Universidad de Duisburg-Essen, Alemania. E-mail: hoppe@collide.info

INTRODUCCIÓN

Las arquitecturas de sistemas distribuidos permiten asignar diferentes servicios en diferentes máquinas dentro de una red. Hoy en día no sólo se usan para aplicaciones de programas de grupos genuinos sino también para proveer servicios orientados hacia usuarios individuales. Una ventaja principal de dichas arquitecturas distribuidas surge a raíz de un acoplamiento débil de los componentes. Esto permite la evaluación del *software* independiente, la combinación flexible de servicios y heterogeneidad en el uso de plataformas de sistemas operativos y lenguajes de implementación.

Desde el punto de vista de sistemas inteligentes, los principios arquitectónicos subyacentes de los sistemas distribuidos comparten muchas características con arquitecturas de agentes. El desarrollo de sistemas basados en dicha arquitectura, para implementar su funcionalidad básica, debe distinguirse con sólo exponer los sistemas existentes totalmente definidos como los servicios para la red. El último, esencialmente sólo requeriría una separación GUI, por el contrario, el primero, influiría en la arquitectura general del sistema y en el enfoque de desarrollo. Hasta ahora estas consideraciones arquitectónicas no han recibido mucha atención del AI en la comunidad educativa. En este sentido, Ritter & Koedinger (1996) sugieren una arquitectura adaptable para agentes de tutorías. Chen (2002) discute el potencial de los servicios en Internet para educación en general con base en la red. En este documento queremos además, explorar el potencial de arquitecturas distribuidas débilmente acopladas para la implementación de ambientes de aprendizaje apoyados inteligentemente con un enfoque particular sobre ambientes de aprendizaje en grupo.

La denominada "arquitectura de pizarrón" es un ejemplo clásico de una arquitectura para sistemas inteligentes que apoyan el acoplamiento débil. En el caso original de HEARSAY (HABLADURÍAS) apoyaba el procesamiento de cascada de la conversación (Erman *et al*., 1980). Desde una perspectiva de sistemas distribuidos, la idea básica del enfoque del pizarrón consiste en reemplazar la comunicación basada en pasar mensajes (entre remitentes y receptores) a publicar información relevante en un espacio accesible por varios agentes o unidades de procesamiento que utilicen operaciones de escritura y lectura sin comunicación directa. Esto le exige a la arquitectura de servidor/cliente un servidor *stateful* (servidor que mantiene datos sobre solicitudes previas para un servicio que luego puede utilizarse en solicitudes posteriores). Para hacer que la información pública se comparta fácilmente sin definiciones de protocolo detalladas es importante depender de unos formatos de datos estructurados en el espacio público. Las arquitecturas de pizarrón pueden entonces implementarse utilizando el enfoque de Espacios *Tuple* (TS, por sus siglas en Inglés), que se origina en el lenguaje Linda de Gelernter (Gelernter, 1985).

Hemos iniciado el uso de SQLSpaces como la implementación de fuente abierta del enfoque TS (Weinbrenner *et al*., 2007), el cual tiene varias ventajas sobre las otras implementaciones existentes. La característica más relevante, en este contexto, es el apoyo para sistemas de lenguajes heterogéneos: las interfaces del cliente para varios lenguajes de programación permiten, por ejemplo, ambientes de aprendizaje interactivo de acoplamiento débil escritas

en Java con componentes de procesamiento basados en *Prolog*. En esta "simbiosis", *Prolog* se utilizaría típicamente para procesamiento más analítico a nivel profundo. Comparado con este enfoque, una arquitectura anterior para ambientes de aprendizaje distribuidos inteligentes (Mühlenbrock *et al*., 1998), combinó el protocolo de ambientes de aprendizaje basados en C++ con componentes de modelación basados en *Prolog* y utilizados para un propósito especial, protocolo de comunicaciones de cableado directo entre los diferentes ambientes de lenguaje. En nuestra solución basada en TS, solamente hay una interfaz *Prolog* de aplicación general sin excluir los formatos de datos específicos. Desde un punto de vista de desarrollo del sistema, no se requieren más interfaces específicas, pero todavía hay una alta flexibilidad al utilizar diferentes formatos de datos. El paradigma de comunicación cambia de pasar mensajes a compartir datos entre componentes débilmente acoplados y posiblemente heterogéneos.

Espacios *Tuple*

El enfoque TS como implementación de la arquitectura de pizarrón fue introducido junto con el lenguaje de coordinación Linda por Gelernter en la década de 1980. La parte principal de este enfoque es un servidor central, que mantiene todos los mensajes. Los clientes únicamente intercambian mensajes con el servidor y no tienen ninguna conexión cliente a cliente. Los mensajes están en un formato *tuple*, es decir, son unas listas ordenadas de campos que contienen datos primitivos. Así que el servidor puede verse como un lugar para intercambiar *tuples*, donde los clientes tienen una memoria de trabajo compartida para resolver problemas de manera paralela. Ya que no hay medios para direccionar a los clientes, pues dos sistemas no pueden explícitamente intercambiar datos entre sí. En cambio, los clientes colocan *tuples* de manera asociativa de acuerdo con su estructura.

En el sistema original se utiliza el lenguaje de coordinación Linda para expresar la semántica de un algoritmo paralelo que utiliza espacios *tuple* como una memoria de trabajo común. Con este propósito Linda incluye tres operaciones básicas out (colocar algo en el servidor), in (tomar algo del servidor) y read (similar a in, pero los deja en el servidor). Para obtener un *tuple* del servidor, una operación in necesita una plantilla denominada argumento. Una plantilla es un *tuple*, cuyos campos pueden ser formales, es decir, no necesitan tener valores, pero también pueden ser comodines escritos (es decir, ("temperatura", "afuera", <flotar>)). Después de recibir dicha plantilla, el servidor trata de encontrar un *tuple* concordante y los devuelve al cliente que consulta.

Implementaciones recientes

Una implementación TS más reciente fue proporcionada como parte del marco Java Jini por Sun Microsystems en 1998 bajo el nombre de JavaSpaces. Fue la primera implementación TS conocida en Java y extendió el marco original de Linda al introducir algunas características nuevas: el tipo de modelo JavaSpaces se adapta a Java, para que en vez de listas de de datos primitivos se tengan que definir clases específicas cuyas variables componentes se interpreten como campos *tuple*. JavaSpaces también proporciona un *mecanismo de evento*

para notificar activamente a los clientes, si ocurre cierto tipos de eventos. Otra extensión de la idea original es el concepto de *leases* (*arrendamiento*). Si después de un tiempo un *tuple* no ha sido actualizado, el servidor puede eliminarlo sin involucrar a ningún cliente. Finalmente un servidor JavaSpaces no es solamente un contenedor *tuple*, sino que consiste en varios espacios disjuntos, que pueden ser tratados por su nombre. Esto le permite a un servidor hospedar varias capas en una aplicación o inclusive varias aplicaciones diferentes en el mismo servidor sin problemas de alteraciones recíprocas. Sin embargo, el JavaSpaces API es menos intuitivo comparado con los primitivos Linda y el uso de Java RMI como un mecanismo de comunicación básica puede resultar en problemas y dificultades adicionales para los programadores que no estén familiarizados con el mismo.

Casi simultáneamente, otra implementación TS basada en Java denominada TSpaces (Lehman *et al*., 1999) se desarrolló y publicó en 1999 por el Centro de Investigación Almaden de IBM. En contraste con JavaSpaces el enfoque principal de este proyecto no fue trasladar la semántica TS al mundo orientado hacia objetos a nivel de estructura de datos (por ejemplo, no tener clases específicas de Java actuando como *tuples*), sino simplemente tener un TS escrito en Java, que es muy similar al sistema Linda original. El manejo de un sistema TSpaces es, desde el punto de vista de un programador, mucho más intuitivo con JavaSpaces, tanto respecto al API como al manejo del servidor. En el lado negativo de TSpaces, hay solamente mecanismos de persistencia muy elementales. Además, las regulaciones de la licencia para TSpaces son mucho más restrictivas respecto al uso y a la redistribución, especialmente para proyectos comerciales.

Diseñamos y desarrollamos SQLSpaces de tal manera que combinara las ventajas de las dos implementaciones TS mencionadas con algunas características adicionales. SQLSpaces usa una base de datos relacional (de ahí su nombre) y proporciona un API limpio y fácil de usar y ofrece características tal como control de versiones, manejo de usuario y de derechos, investigación en tiempo real a través de Internet y heterogeneidad del lenguaje, además de las características tradicionales descritas anteriormente. Especialmente las interfaces del cliente para lenguajes de programación bastantes diferentes (Java, C#, PHP, Ruby, *Prolog*) hace de esta una opción interesante para construir arquitecturas heterogéneas. Por lo tanto, decidimos utilizar SQLSpaces como la plataforma básica para nuestro proyecto.

Aplicaciones relevantes

Aunque no es muy nueva, la idea TS recientemente ha recibido atención en el diseño y la implementación de ambientes cooperativos: Group Scribbles (Brecht *et al*., 2006) es un ambiente cooperativo para convertir notas gráficas y de texto ("scribbles" (garabatos)). Group Scribbles fue desarrollado en el marco del Proyecto de Espacios *Tuple* en el Centro para Tecnología en el Aprendizaje en SRI International. Group Scribbles utiliza TSpaces para sincronización (Roschelle *et al*., 2005) y está disponible para plataformas diferentes. Desde un punto de vista arquitectónico, Group Scribbles no tiene una capa de sincronización y comunicación de aplicación independiente. Por lo tanto, no es posible reutilizar el código para sincronizar otras aplicaciones distribuidas. Sin embargo, en nuestro enfoque la aplicación del

cliente utiliza una capa de *middleware* (*software* especializado) independiente que puede utilizarse para clientes diferentes y puede adaptarse a varias implementaciones TS sin cambiar la aplicación anterior.

Una segunda aplicación relevante fue desarrollada en el proyecto *Amenities project* (Garrido *et al*., 2006). Este utiliza JavaSpaces con el propósito de sincronizar libros de citas de personas en áreas de investigación académica. Un enfoque principal de este estudio de caso está en la provisión de funciones valiosas de conocimiento utilizando la arquitectura TS. Ni esta aplicación ni Group Scribbles gestionan la heterogeneidad del lenguaje en términos de acceder a TS usando lenguajes de programación diferentes.

El escenario de aplicación

A continuación, tratamos ambientes de discusión cara a cara en donde una pantalla interactiva grande sirve como punto común de referencia para exhibir aportes diferentes relacionados entre sí tal como se describe en la Figura 1. En este escenario, los PDAs sirven como dispositivos de entrada, es decir no se utilizan para replicar la gráfica de la discusión (cf. Bollen *et al*., 2006) sino que los estudiantes los utilizan para aportar a la discusión. En el nivel de implementación, la aplicación comprende tres tipos diferentes de componentes del sistema escritos en diferentes lenguajes: un ambiente de usuario final para corridas de participantes / colaboradores en los PDA y se escribe en C#; la pantalla interactiva grande está ocupada por un ambiente de modelación visual basado en Java (denominado *FreeStyler*, ver Hoppe & Gassner, 2002); los agentes inteligentes que analizan discusiones en curso se definen en *Prolog*.

Las siguientes características basadas en la arquitectura y en el enfoque TS resultaron específicamente relevantes y beneficiosas:

- Una vez que se suministra la interfaz TS del cliente para cierto lenguaje, no se requieren más interconexiones sintácticas para una aplicación específica.

- El patrón de comunicación de pizarrón permite un alto grado de independencia (acoplamiento débil) en el desarrollo de los diferentes componentes del sistema.

- La heterogeneidad del lenguaje apoya una forma efectiva de especialización y distribución del trabajo en un equipo de programación.

Figura 1. El escenario de aplicación

Una versión específica del último punto es que, para definir los patrones de colaboración, un ingeniero del conocimiento puede directamente interactuar con el ambiente bajo desarrollo sin tener que tratar problemas de interconexión de nivel bajo. Estas ventajas se explicarán de manera más detallada en el resto del documento con base en la implementación existente.

Motivación

El escenario original de *MobileNotes* se describe en Bollen *et al*., (2006): facilita discusiones cara a cara en aulas o salones de reuniones en donde los PDA no se utilizan para compartir información relevante sino para proporcionar entradas individuales en forma de notas o bosquejos pequeños. En este escenario, la información para compartir por el grupo de discusión se exhibe en una pantalla interactiva grande o en una pizarra virtual como un espacio de referencia común. Este diseño fue respaldado por hallazgos de Liu & Kao ((2005), que corroboran las ventajas de un foco común de atención visual en discusiones de grupo. Ahora, hemos modificado el escenario bastante general de *Mobile Notes* para manejar una representación estructurada más específica del contenido de la discusión. Esta representación (QOC) tiene su origen en el área de métodos de *"design rationale"* (racionamiento del diseño) (MacLean *et al*., 1991) y apoya la externalización de un proceso de decisión en el diseño.

El hecho de tener una aplicación más especializada y una representación más estructurada que la del escenario *Mobile Notes* nos permite analizar el contenido de la discusión y procesar información basada en ciertos patrones temporales y estructurales. En este contexto, el modelo de comunicación por pizarrón se utiliza para enchufar los agentes analíticos correspondientes.

Decisión del diseño basada en QOC

El concepto de "racionalización del diseño" comprende una variedad de métodos para apoyar la toma estructurada de decisiones y la externalización en el diseño de artefactos (técnicos) (cf. Regli *et al*., 2000). Las técnicas de racionalización del diseño fueron aplicadas y desarrolladas particularmente en el campo del GUI y del diseño de interacción. Las tecnologías de soporte para los enfoques de racionalización del diseño incluyen la provisión de representaciones legibles por máquina (para ser compartidas por humanos) así como arreglos de medios interactivos/cooperativos para facilitar la comunicación y el flujo en el proceso del diseño.

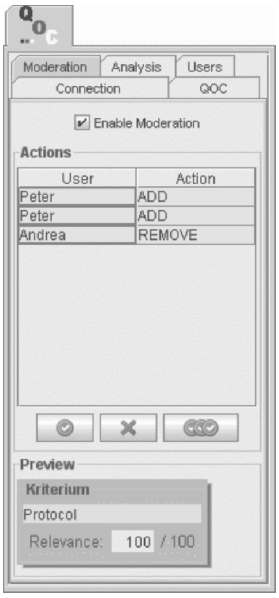

Figura 2. Interfaz de moderación con vista previa de la acción

"Preguntas — Opciones — Criterios" o QOC (por sus siglas en inglés) (MacLean *et al*., 1991) es, en primer lugar, una representación estructurada para utilizar en el proceso de racionamiento del diseño, comenzando con una pregunta o un tema de diseño específico para ser resolver, luego enunciando opciones alternativas e introduciendo criterios para una evaluación comparativa y ponderada de la opciones. El producto final es un gráfico QOC que documentaría y explicaría la decisión específica (ver Figura 5).

La representación QOC está apoyada por una "palette" (pizarrón de colores) específico en nuestro ambiente de modelación visual cooperativo *FreeStyler* (Hoppe & Gassner, 2002). Se ha utilizado con bastante éxito en los salones de clase así como en proyectos de desarrollo de *software* por los estudiantes. El escenario de aplicación específico para apoyar aquí es el uso de QOC en un escenario cara a cara en donde la representación visual compartida se exhibe en un

tablero interactivo bajo control de un profesor o un moderador. Los participantes utilizan los PDAs para hacer entradas a este modelo compartido. Los resultados de los agentes analíticos generan retroalimentación a la pantalla pública como un tipo de contenido de información de conocimientos orientada al contenido.

Discusión, moderación

Adicionalmente, la arquitectura descrita en el siguiente capítulo permite la moderación que venga de los aportes de los estudiantes.

En este documento, el término moderación describe la posibilidad de un moderador o un profesor para ver con antelación y aprobar o rechazar las acciones del estudiante. Las acciones singulares en nuestro caso son:

- agregar una pregunta, opción o criterio
- eliminar o cambiar el contenido de una de las anteriores
- agregar o quitar un borde
- cambiar el peso de un borde

Si una de estas acciones las realiza un estudiante, aparecerá en el interfaz de moderación en el ambiente *FreeStyler* que está bajo control del moderador (ver Figura 2). Aquí, todas las acciones se recopilan en una lista y se pueden ver de antemano. El moderador puede optar por aprobar una acción (es decir, la acción se ejecuta en el StateSpace (ver capítulo siguiente)) o la puede rechazar (es decir, la acción no se ejecuta y se quitará del panel del moderador). Claro está, que la característica de moderación puede apagarse totalmente. En este caso, todas las acciones de los estudiantes se ejecutan directamente sin interferencia o retraso adicional.

Arquitectura general

La arquitectura general consiste en que la aplicación QOC-*FreeStyler* se muestra en la pantalla pública, varios clientes pueden correrla en PDAs, un análisis de motor (*Prolog*) y un servidor *Tuple* Space (TS) como plataforma de comunicación y sincronización (ver Figura 3).

EL TS consiste de varios espacios que juegan un papel central en la arquitectura. El StateSpace mantiene una representación del modelo QOC actual que se visualiza por la aplicación *FreeStyler*. Los clientes PDA pueden realizar acciones para modificar o agregar al modelo, que serán escritas dentro del primer *ActionSpace*. En ese punto, tal como se describió en el capítulo anterior, las acciones pueden moderarse (es decir, verse con antelación y posiblemente aprobadas por un moderador) o realizarse de inmediato. La separación de las acciones y los estados no solamente permiten una implementación fácil de dicha moderación, sino que proporciona una forma conveniente para el análisis de la acción, ya que todas las acciones están continuamente disponibles en el motor *Prolog* (ver capítulo siguiente).

El motor de análisis trata continuamente de coincidir los patrones dados a los contenidos de *StateSpace* y *ActionSpace*. Si un patrón coincide, el resultado se escribe en el *AnalysisSpace*, desde donde puede recuperarse y visualizarse por FreeStyler.

Ingeniería del conocimiento de patrones de cooperación

La Figura 4 muestra el proceso de ingeniería del conocimiento de patrones de cooperación. Un ingeniero del conocimiento utiliza el ambiente de pruebas *Prolog* para crear y probar predicados *Prolog* que coincidan con patrones interesantes en la acciones del usuario o en estados de modelos (o inclusive en una mezcla de los dos). Una vez que el ingeniero del conocimiento termine el proceso de creación del patrón, se anotan los predicados con las descripciones sobre el significado de los patrones y de los parámetros utilizados. Esta información la utiliza posteriormente el *Awareness Display* para transmitir información sobre patrones encontrados para el usuario final en el momento de la corrida (ver Figura 4).

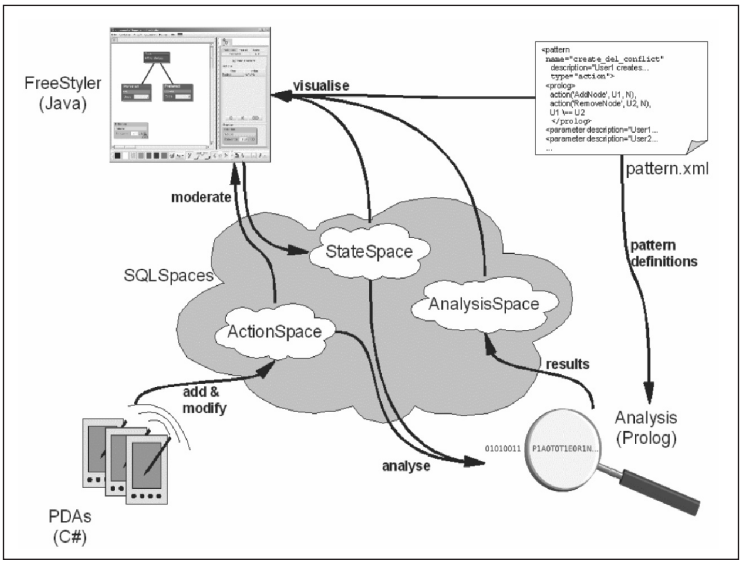

Figura 3. Arquitectura general

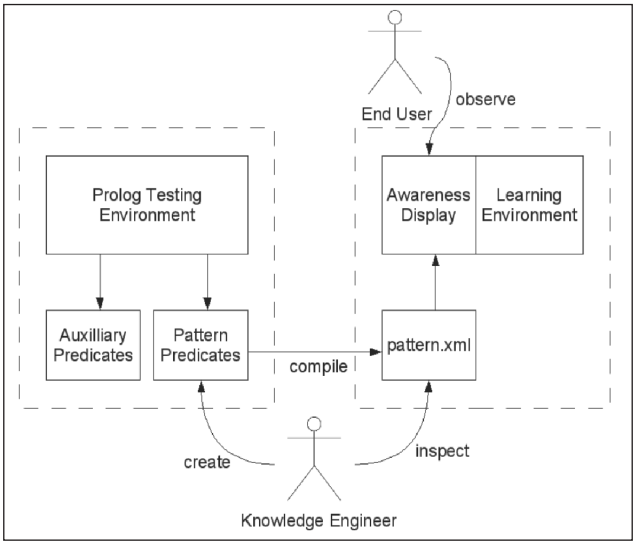

Figura 4. Ingeniería del conocimiento de patrones de cooperación

Las siguientes líneas son un extracto del archivo pattern.xml que usada por el A*warness Display* (Pantalla del Conocimiento).

```
<pattern
  name="create_del_conflict"
  description="User1 creates a node and a different User2
                deletes it"
  type="action">
  <prolog>
   action('AddNode', U1, N),
   action('RemoveNode', U2, N),
   U1 \== U2
  </prolog>
  <parameter description="User1" var="U1"/>
  <parameter description="User2" var="U2"/>
  <parameter description="Node" var="N"/>
</pattern>
```

Una ventaja de este tipo de marco de ingeniería del conocimiento es la homogeneidad de la interfaz. En este enfoque, el ingeniero del conocimiento puede crear, probar y describir patrones de análisis en un solo ambiente (es decir, un ambiente de programación *Prolog* similar a SWI-*Prolog*).[10] Aquí, todo el desarrollo interactivo y el proceso de pruebas ocurren en el ambiente de trabajo familiar. Además, al tener el archivo pattern.xml, se pueden

10 Ver http://www.swi-*prolog*.org, 13 de Noviembre de 2009

intercambiar los patrones de análisis, sus descripciones así como los metadatos entre usuarios y moderadores simplemente al copiar un archivo.

Patrones de acción y patrones de estado

Tal como se dijo anteriormente, el marco descrito permite un análisis de desempeño basado en información sobre el estado real de un modelo así como análisis basados en la historia de las acciones del usuario. Típicamente, *state patterns* (patrones de estado) tratan sobre características tácticas y semánticas y características del lenguaje de modelación utilizado. Estos tipos de patrones describen ciertas constelaciones de nodos, bordes y sus atributos. Para el método QOC, identificamos algunos patrones que son potencialmente interesantes para un moderador:

- El modelo contiene un criterio desconectado;
- el modelo contiene una opción sin criterios adjuntos;
- no hay opción preferida en el modelo;
- un criterio tiene efectos iguales para todas las opciones;
- ... y otros.

Action patterns (Patrones de acción) se ocupan de ciertas secuencias en la historia de las acciones del usuario. Típicamente, estos patrones describen algún tipo de comportamiento significativo y posiblemente interesante de usuarios individuales o describen ocurrencias de colaboración entre dos o más usuarios. Estos patrones tienden a ser más independientes en el dominio que los patrones de estado, así que pueden utilizarse potencialmente con varios lenguajes de modelación y aplicaciones de aprendizaje.

Ejemplos para patrones de acción son:

- Un usuario crea un objeto, un usuario diferente elimina este objeto (¿conflicto?)
- Un usuario crea un objeto, un usuario diferente conecta este objeto (¿colaboración?)
- Un usuario claramente realiza la mayoría de las acciones (¿domina a otros?)

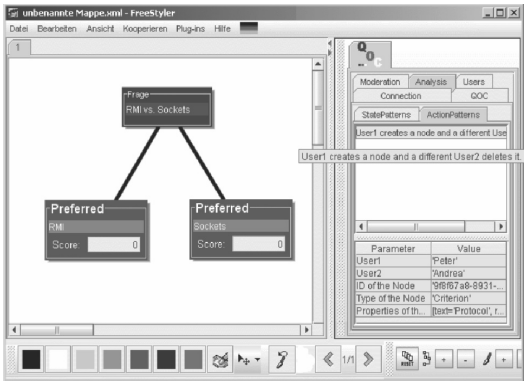

Figura 5. Modelo QOC y pantalla del conocimiento para crear / eliminar conflicto

Sin embargo, hay patrones de acción dependientes del dominio, es decir, para QOC, una serie de acciones que cambian el peso de un borde de valores positivos a negativos puede ser muy significativa y podría no ser detectada solamente por patrones de estado.

Adicionalmente, las combinaciones de patrones de estado y de acción son factibles y significativas, también. Imagínese que tiene un patrón que detecta una situación característica en el modelo (análisis de estado) en combinación con la búsqueda de posibles acciones colaboradoras de usuarios que lleven a esta situación (análisis de acción).

Perspectiva

En desarrollos posteriores, planeamos generalizar y extender este enfoque y la arquitectura a varios lenguajes de modelación, incluyendo Petri Nets, System Dynamics o UML. Se pueden identificar predicados *Prolog* predefinidos que son útiles y significativos para análisis de estado y análisis de acciones en varios lenguajes en este proceso de generalización. Se utilizarán agentes adicionales para procesar aún más el contenido del *Analysis Space* para generar retroalimentación directa en forma de recomendaciones o adaptaciones del ambiente (en vez de solo exhibir la información del conocimiento).

Además, planeamos incluir las acciones en la cola de moderación (es decir, acciones que han sido realizadas por usuarios pero no han sido aún aprobadas por un moderador) dentro del ciclo de análisis. Por lo tanto, un moderador podría ser capaz de evaluar el impacto de las acciones del usuario antes de publicarlas.

REFERENCIAS BIBLIOGRÁFICAS

BOLLEN, L.; JUAREZ, G.; WESTERMANN, M.; HOPPE, H.U. (2006). PDAs as input devices in brainstorming and creative discussions. Proceedings of 4th International Workshop on Wireless, Mobile and Ubiquitous Technologies in Education (WMUTE 2006). Los Alamitos, CA 2006 (IEEE Computer Society), pp. 137-141.

BRECHT, J.; PATTON, C.; CHAUDHURY, S. R.; DIGIANO, C.; TATAR, D.; ROSCHELLE, J.; DAVIS, K. (2006). Coordinating networked learning activities with a general-purpose interface. Proceedings of mLearn 2006, Banff (Canada), October 2006.

CHEN W. (2002). Web Services - What Do They Mean to Web-based Education? Proceedings of ICCE 2002, Auckland, New Zealand, December 2002. pp. 707–708.

ERMAN, L. D.; HAYES-ROTH, F.; LESSER, V. R.; REDDY, D. R. The Hearsay-II Speech-understanding System: Integrating Knowledge to Resolve Uncertainty. ACM Computing Surveys, 12(2):213–253, 1980.

GARRIDO, J. L.; NOGUERA, M.; GONZÁLEZ, M.; GEA. M.; HURTADO, M.V. (2006). Leveraging the Linda coordination model for a groupware architecture implementation.

Proceedings of 12th International Workshop on Groupware (CRIWG 2006), Medina del Campo, Spain, Sept. 2006. pp. 286–301.

GASSNER, K.; JANSEN, M.; HARRER, A.; HERRMANN, K.; HOPPE, H.U. (2003). Analysis methods for collaborative models and activities. In Proceedings of CSCL 2003, Bergen (Norway), June 2003. pp. 369-377.

GELERNTER, D. (1985). Generative communication in Linda. ACM Trans. Program. Lang. Syst., 7(1):80–112.

HOPPE, H.U.; GASSNER, K. (2002). Integrating collaborative concept mapping tools with group memory and retrieval functions. In Proceedings of CSCL 2002, Boulder (USA), January 2002. pp. 716-725.

LEHMAN, T.J.; MCLAUGHRY, S.W.; WYCKOFF, P. (1999). T Spaces: The Next Wave. Proceedings of HICSS 1999, Maui, Hawaii, USA, Jan. 1999.

LIU, C.; KAO, L. (2005). Handheld devices with large shared display groupware: tools to facilitate group communication in one-to-one collaborative learning activities. In Proceedings of IEEE WMTE 2005, Tokushima, Japan, March 2005. pp. 128-135.

MACLEAN, A.; YOUNG, R.; BELLOTI, V.; MORAN, T. (1991). Questions, options, and criteria: Elements of design space analysis. Human-Computer Interaction, 6 (3/4), 201–250.

MÜHLENBROCK, M.; TEWISSEN, F.; HOPPE, H.U. (1998). A framework system for intelligent support in open distributed learning environments. International Journal of Artificial Intelligence in Education, vol. 9, 256-274.

REGLI, W.C.; HU, X.; ATWOOD, M.; SUN, W. (2000). A survey of design rationale systems: Approaches representation, capture and retrieval. Engineering with Computers 16: 209–235.

RITTER, S.; KOEDINGER, K. (1996). An Architecture For Plug-In Tutor Agents. International Journal of Artificial Intelligence in Education 7 (3/4): 315-347.

ROSCHELLE, J., SCHANK, P., BRECHT, J., TATAR, D., & CHAUDHURY, S. R. (2005). From response systems to distributed systems for enhanced collaborative learning. Proceedings of ICCE 2005, Seoul, Korea, Nov./Dec. 2002. pp. 363-370.

WEINBRENNER, S.; GIEMZA, A.; HOPPE, H.U. (2007). Engineering heterogeneous distributed learning environments using TupleSpaces as an architectural platform. in Proceedings of the 7th IEEE International Conference on Advanced Learning Technologies (ICALT 2007). Los Alamitos (USA): IEEE Computer Society, 2007. pp. 434-436.