APROXIMACIÓN PARA ABORDAR PROBLEMAS DE ASIGNACIÓN DE RECURSOS

- UN MODELO GENERAL ORIENTADO A OBJETOS -

Susana Chavez, Silvina Migani
e-mail: {schavez,smigani}@iinfo.unsj.edu.ar

Docentes del Departamento de Informática
Investigadores del Instituto de Informática
LISI
Universidad Nacional de San Juan
Cereceto y Meglioli (5400) San Juan, Argentina

1. INTRODUCCIÓN

Existen problemas del mundo real que por su complejidad presentan una explosión combinatoria en su tratamiento. Entre ellos podemos detallar los problemas industriales como planificación, scheluding, optimización, asignación de recursos, etc.

Los problemas a los que está orientado este trabajo en particular son de asignación de recursos.

En este trabajo se propone una aproximación a la modelización orientada a objetos para el tratamiento general de este tipo de problemas. [1]

Un conjunto particular de recursos se representa como una clase de objetos del mundo real que incluye atributos ocultos y métodos encapsulados. Los componentes se comunican a través de mensajes, brindando inmunidad al resto del sistema frente a cambios de implementación.

Este modelo general además de representar más naturalmente estos problemas, hace posible la incorporación de nuevas funcionalidades sin que tenga un impacto sustancial.

Al momento de la implementación este modelo resulta fácilmente tratable utilizando un lenguaje de programación por restricciones (Constraint Programing).

[1] Este trabajo está enmarcado dentro del Proyecto de Investigación, aprobado por CICITCA, denominado "Resolución de Problemas de Asignación de Recursos usando Técnicas de Inteligencia Artificial".

2. PROBLEMÁTICA A TRATAR

Los problemas de asignación de recursos, denominados de combinatoria discreta, son aquellos que tienen tal complejidad que no pueden ser resueltos eficientemente o pueden requerir demasiado tiempo de ejecución usando métodos tradicionales de la programación convencional. Uno de los aspectos más importantes de este enfoque es que el modelo teórico propuesto se extiende a distintos problemas de la realidad que involucran asignación de recursos, tales como:

- Conductores de una empresa de transporte que deben ser asignados a unidades en distintos turnos
- Profesores que deben dictar clases en aulas y horarios diferentes durante los cinco días de la semana
- Máquinas que deben utilizar diferentes recursos en el proceso de producción
- Cajeras de un supermercado que deben ser asignadas a una de las diferentes cajas en los distintos turnos.

En general, existen entidades que tienen que ser asignadas a lugares y también se presentan restricciones que limitan tal asignación. Esto conduce a una explosión combinatoria, que se resuelve eficientemente aplicando técnicas de programación con restricciones (Constraint Programming - CP). En este marco han surgido desarrollos muy importantes [Leler,90] [Ibañez,94] [Forradellas,94] [Rueda,95], principalmente en Constraint Logic Programming (CLP) para dominios finitos, tales como CHIP (Constraint Handling In Prolog) [Dincbas,88] [Van Hentenryck,89] [Forradellas,91].

La integración de la programación con restricciones con la programación orientada a objetos (POO), que facilita la comprensión, resolución y extensión de problemas, permite resolver eficientemente un amplio rango de problemas de combinatoria discreta, principalmente aquellos de complejidad no polinomial, intratables con métodos tradicionales.

En este paper, se propone un modelo general orientado a objetos que representa los problemas de combinatoria de asignación de recursos, los cuales se resuelven integrando la CP y la POO.

3. FORMALIZACIÓN DEL PROBLEMA

3.1. Modelo Teórico General

Un problema de satisfacción de restricciones (Constraint Satisfaction Problems - CSP) está compuesto por:

1. un conjunto de variables, $V_{1, \ldots, } V_n$,
2. un conjunto de dominios, $D_{1, \ldots, } D_n$, asociado a las variables,
3. un conjunto de restricciones relativas a las variables.

Una solución es un conjunto de valores $a_{1, \ldots, } a_n$, donde $a_j \in D_j$ y la sustitución a_j / V_j verifica todas las restricciones ($1 \leq j \leq n$).

La obtención de la solución consiste en explorar el espacio de soluciones guiada mediante el conjunto de restricciones. las técnicas de resolución más utilizadas consisten en la expansión de un árbol de búsqueda que considere todas las opciones. Se realiza en cada nodo del árbol un proceso de comprobación de consistencia considerando las asignaciones realizadas hasta ese momento. Si las asignaciones resultan inconsistentes, **no** debe considerarse el subárbol correspondiente produciendo una poda *a priori* del espacio de búsqueda. De esta manera se van eliminando valores de los dominios de las variables hasta encontrar los valores que dan solución al problema.

A partir de un CSP general se define un nuevo modelo que trata problemas de Combinatoria de Asignación de Recursos.

3.2. Modelo Teórico Propuesto

3.2.1. El Espacio CSP

Un CSP de Asignación de Recursos, llamado *espacio CSP* consiste de los siguientes componentes:

1. un conjunto de variables $Vi_1,...,i_k$

   donde $1 \leq i_1 \leq l_1$,

   $1 \leq i_k \leq l_k$.

2. un único dominio D asociado a las variables, y

3. un conjunto de restricciones lineales que relacionan a las variables.

Una solución es un conjunto de valores $a_{i1},...,_{ik} \in D$, tal que la sustitución $a_{i1},...,_{ik}$ / $Vi_1,...,i_k$ verifica todas las restricciones ($1 \leq i_1 \leq l_1$, ..., $1 \leq i_k \leq l_k$).

En el CSP convencional, las variables pueden ser vistas como puntos en un segmento, mientras que en el Espacio-CSP las variables pueden ser vistas como puntos de un espacio n-dimensional denotado como S.

Un *grupo*, simbolizado como G, se define como un subconjunto del conjunto de variables $Vi_1,...,i_k$, donde $a_j \leq i_j \leq b_j$ ($1 \leq j \leq k$), con $1 \leq a_j$ y $b_j \leq l_j$. El par $< a_j,b_j >$ representa el rango del $j^{ésimo}$ índice de la variable $Vi_1,..., Vi_k$.

Luego, una restricción puede asociarse a uno o a varios grupos, dependiendo del tipo de restricción, como se muestra a continuación. En el mundo real las variables representan lugares a ser asignados, mientras que D representa el conjunto de todos los posibles candidatos a ocupar estos lugares. De ahora en más, los elementos de D son llamados *candidatos*.

3.2.2. Tipos de Restricciones

Dado un *grupo* de puntos G, definiremos restricciones de diferentes tipos. Diremos que las restricciones involucran puntos y candidatos con *peso* 0 si los candidatos "no deben" ser asignados a esos puntos, y con *peso* 1 si los candidatos "deben" ser asignados a los puntos. Los *pesos* son denotados como *W*.

A continuación describimos diferentes tipos de restricciones correspondientes a varios problemas de combinatoria discreta de Asignación de Recursos:

Tipo1: Algunos candidatos pueden ocupar un grupo de puntos con peso 0.

Tipo2: Cualquier candidato que puede ocupar algún *punto* de un *grupo* con peso 0 o 1, puede ocupar otro *punto* del mismo grupo con peso 0 o 1.

Tipo3: En un grupo determinado, el candidato i puede ocupar un *punto* con peso 0 o 1, y el candidato j puede ocupar otro punto con peso 0 o 1.

Tipo4: Un candidato puede ocupar un punto de un *grupo* con peso 1.

Tipo5: Un candidato que puede ocupar un punto de un *grupo* con peso 0 o 1, puede ocupar cualquier punto de otro *grupo* con peso 0 o 1.

Las restricciones enumeradas previamente son generales para muchos problemas de Asignación de Recursos. Sin embargo, pueden existir otras restricciones que podrían ser clasificadas e incluidas en algunos de los tipos definidos previamente, o bien generar otras clases que debieran incluirse en el modelo.

4. MODELAMIENTO

4.1. TÉCNICA DE MODELAMIENTO DE OBJETOS (TMO)

El modelamiento y diseño orientado a objetos promueve un mejor conocimiento de los requerimientos, diseños más claros y sistemas más mantenibles.

Dado un problema a resolver, la orientación a objetos brinda una forma más abstracta de pensar, no en términos computacionales. Brinda una forma más práctica y productiva de desarrollo de software, independientemente del lenguaje de programación en que se implemente.

Un sistema construido usando la tecnología de orientación a objetos es una colección de objetos cooperantes con identidad propia. Un objeto es una entidad que tiene un conjunto de responsabilidades y que encapsula un estado interno. Las responsabilidades de un objeto (las operaciones que puede realizar) se implementan mediante métodos, los cuales describen el comportamiento del mismo. El estado interno se implementa mediante un conjunto de atributos encapsulados.

Los objetos que comparten idénticas propiedades (atributos y comportamiento) se agrupan en clases. Una clase contiene la descripción de las características comunes de todos los objetos que pertenecen a ella, es decir, la especificación del comportamiento, la definición de la estructura interna y la implementación de los métodos. De esta manera, un objeto es una instancia de una clase.

La principal ventaja del desarrollo orientado a objetos no sólo es facilitar la reusabilidad en una aplicación, sino ofrecer la posibilidad de reusar diseño y código en proyectos futuros. Algunas de las herramientas que lo permiten son la abstracción, la encapsulación, la herencia y el polimorfismo.

Abstracción: Consiste en considerar lo esencial, los aspectos relevantes de una entidad, ignorando las demás propiedades. En el desarrollo de software, significa focalizarse en lo que un objeto es y hace, antes de decidir cómo será implementado. De esta manera, el uso de la abstracción durante la etapa de análisis significa tratar con conceptos del dominio de la aplicación, sin tener que tomar decisiones de diseño e implementación antes de que el problema sea totalmente conocido.

Encapsulación: Como dijimos, un objeto es una entidad que encapsula un estado interno. Por lo tanto, la encapsulación permite separar los aspectos externos de un objeto (accesibles por otros objetos), de los detalles de implementación (ocultos para el resto).

Herencia: Muchas veces encontramos que clases diferentes poseen propiedades comunes. El mecanismo de abstracción generalización-especialización consiste en construir jerarquías de clases-subclases, es decir, es una relación entre una clase y una o más versiones refinadas de ella. La clase que se está refinando se llama superclase y cada versión refinada se llama subclase, donde éstas heredan todas las propiedades de la superclase y, a su vez, pueden agregar propiedades específicas. Es decir, el término herencia refiere al mecanismo por el cual, a través de una relación de generalización diferentes clases comparten propiedades. Por consiguiente, las subclases heredan en particular el comportamiento de la superclase, es decir, sus operaciones. Estas operaciones pueden ser redefinidas para que se comporten diferente de acuerdo a cada subclase. Esta característica fundamental de la orientación a objetos se denomina polimorfismo.

Polimorfismo: Primero recordemos que la implementación de una operación en una clase se denomina método. El polimorfismo significa que la misma operación puede comportarse de forma diferente aplicada a objetos de clases diferentes. Cada objeto conoce cómo responder a sus responsabilidades. Por lo tanto, el usuario de una operación, no necesita conocer que métodos existen para implementar una operación polimórfica. Es más, pueden agregarse nuevas clases que provean nuevas implementaciones sin necesitar cambiar el código existente.

La orientación a objetos ofrece además otras ventajas, como son:

- una reducción de la brecha semántica: Es decir, permite generar un modelo más natural. Preserva la semántica del dominio del problema.
- incrementa la consistencia entre el análisis, el diseño, y la programación.
- separación del qué y del cómo.
- mayor modularidad
- facilita la prototipación
- brinda un efecto limitado frente a cambios
- aumenta la productividad: Se ha estimado que el 80 por ciento de los costos dedicados a la producción de software se produce después de su desarrollo inicial; esto significa que muchos defectos se descubren mas tarde, y mas importante aun, mucha de la funcionalidad del sistema se alcanza después del desarrollo inicial. De esta manera, un método que enfatiza el mantenimiento, mejora sustancialmente la productividad.
- extensibilidad: Los conceptos propios de la orientación a objetos hacen posible que el incorporar nuevas funcionalidades a un sistema tenga un impacto sustancialmente menor que si se hubiera desarrollado usando otra metodología.
- comunicación a través de mensajes: Los objetos componentes del sistema se comunican a través de mensajes (son invocaciones a operaciones), brindando inmunidad al resto del sistema frente a cambios de implementaron, obviamente siempre y cuando la interface no sea alterada. Este concepto esta totalmente

vinculado con la encapsulación, ya que los detalles de implementación están encapsulados dentro del objeto receptor.

4.2. MODELO TEÓRICO GENERAL

Tomando el modelo teórico del Espacio_CSP y los tipos de restricciones definidos, construimos un modelo orientado a objetos del sistema (figura 1), basado en la UML (Unified Modeling Languaje). Las componentes del espacio y los tipos de restricciones son representados como objetos. Este modelo describe el comportamiento y la estructura de los objetos del sistema: su identidad, su estructura interna y sus relaciones con otros objetos.

Analizando el modelo generado se puede observar que un espacio es un objeto compuesto que contiene otros objetos:

a) ejes,

b) candidatos,

c) grupos y

d) restricciones

La cantidad de ejes de un espacio está dada por la dimensión del mismo. Los ejes, a su vez incluyen valores. Una vez definidos los ejes y los valores de ejes, quedan establecidos los puntos que son las variables del espacio.

Los candidatos representan el dominio asociado a las variables.

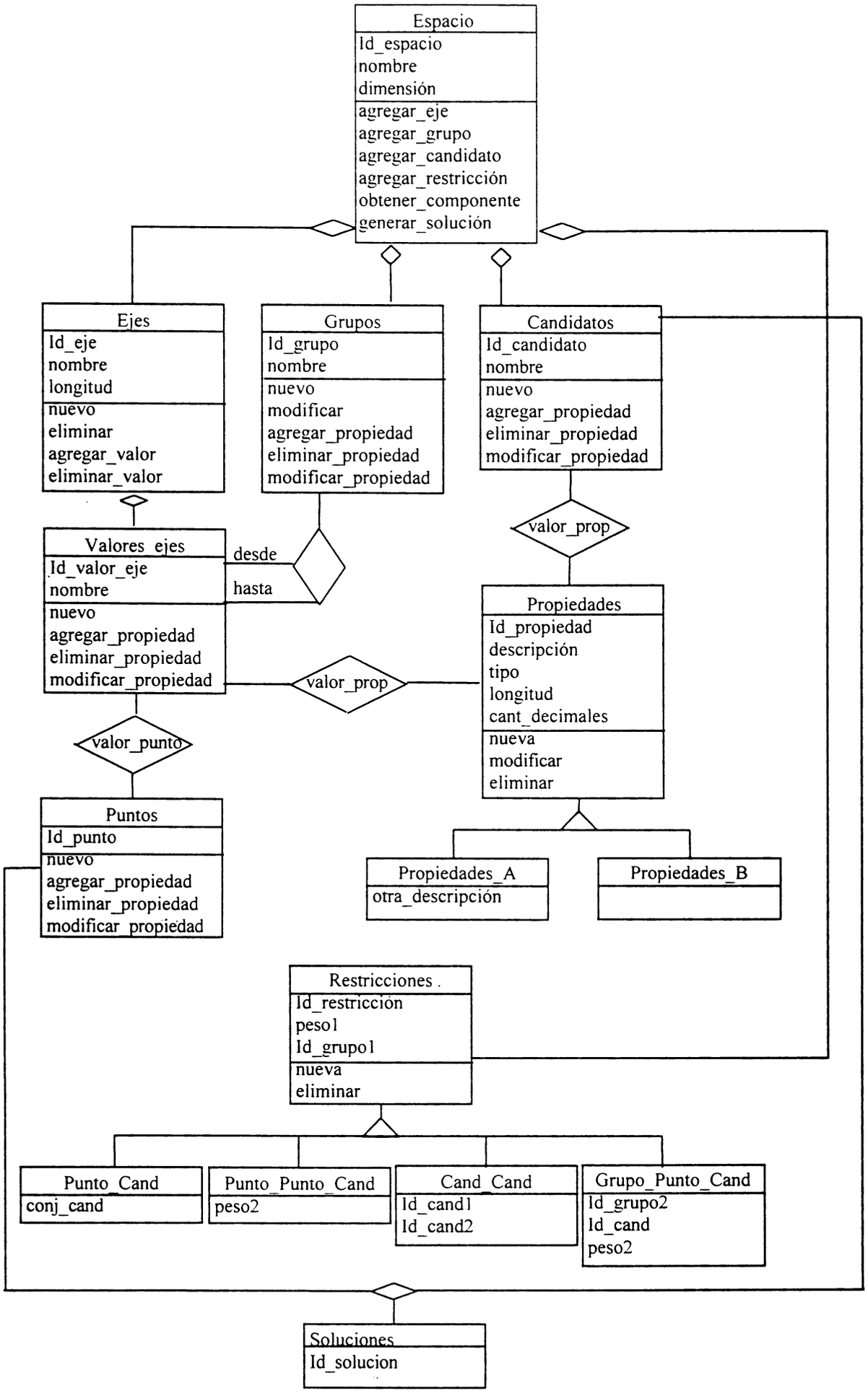
Los grupos se definen en función a valores de ejes. Cumplen un papel importante ya que sobre ellos se aplicaran las restricciones.

Las restricciones limitan la asignación de valores a las variables dentro de una solución. Se puede notar que existe una clase genérica Restricciones refinada en 4 subclases. Los tipos 1 y 4 son unificados puesto que comparten la misma estructura.

A su vez, tanto los candidatos como los valores de ejes pueden tener propiedades, como por ejemplo, edad de un candidato, hora de inicio y finalización de un turno, etc. Estas propiedades también son modeladas como objetos. Existen dos tipos de propiedades, tipo A y tipo B. Las propiedades tipo A, responden a propiedades distintas de candidatos y valores de ejes que son comparables entre si. Por ejemplo, la propiedad capacidad de un aula (propiedad de un valor de eje) y cantidad de alumnos de un profesor (propiedad de un candidato), pueden necesitar compararse. Las propiedades del tipo B son comparables solo con propiedades idénticas o bien con constantes.

Las soluciones también son representadas como objetos, que vinculan los puntos del espacio con los candidatos. Donde toda solución verifica todas las restricciones definidas para dicho espacio.

Vale la pena aclarar que el modelo orientado a objetos propuesto es el punto de partida en la resolución de un problema de asignación de recursos especifico, a partir del cual se pueden definir nuevas clases o bien redefinir algunas de las existentes.


Espacio
Id_espacio
nombre
dimensión
agregar_eje
agregar_grupo
agregar_candidato
agregar_restricción
obtener_componente
generar_solución
Ejes
Id_eje
nombre
longitud
nuevo
eliminar
agregar_valor
eliminar_valor
Grupos
Id_grupo
nombre
nuevo
modificar
agregar_propiedad
eliminar_propiedad
modificar_propiedad
Candidatos
Id_candidato
nombre
nuevo
agregar_propiedad
eliminar_propiedad
modificar_propiedad
valor_prop
Valores_ejes
Id_valor_eje
nombre
nuevo
agregar_propiedad
eliminar_propiedad
modificar_propiedad
desde
hasta
valor_prop
Propiedades
Id_propiedad
descripción
tipo
longitud
cant_decimales
nueva
modificar
eliminar
valor_punto
Puntos
Id_punto
nuevo
agregar_propiedad
eliminar_propiedad
modificar_propiedad
Propiedades_A
otra_descripción
Propiedades_B
Restricciones
Id_restricción
peso1
Id_grupo1
nueva
eliminar
Punto_Cand
conj_cand
Punto_Punto_Cand
peso2
Cand_Cand
Id_cand1
Id_cand2
Grupo_Punto_Cand
Id_grupo2
Id_cand
peso2
Soluciones
Id_solucion

5. APLICACIÓN A UN PROBLEMA REAL

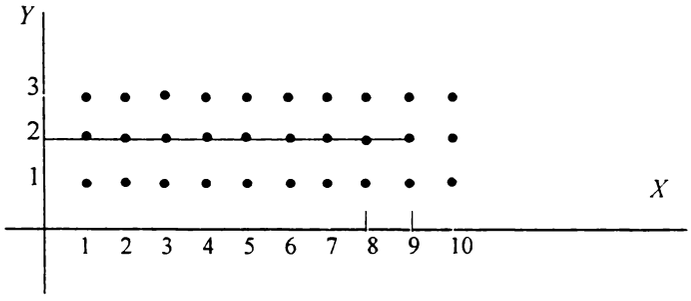
Una Empresa de Transporte de pasajeros cuenta con ***n*** unidades, y ***m*** conductores que deben ser asignados a las unidades en uno de los tres turnos estipulados (Mañana: 6:00 -13, Tarde: 13:00 - 20:00 y Noche: 20:00 - 3:00), teniendo en cuenta que se deben satisfacer las condiciones impuestas por la empresa.

Este problema puede ser modelado como un Espacio-CSP. Para ello es importante, en primer lugar, identificar los ejes. La cantidad de ejes determina la dimensión del espacio.

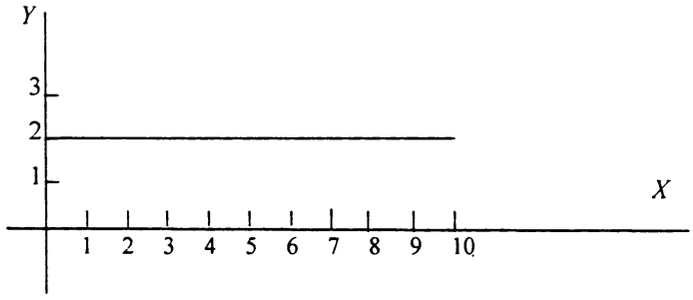
Como procedimiento general, proponemos identificar las entidades más importantes del problema. En este caso, son: las unidades, los conductores y los distintos turnos. Los conductores son los Candidatos y las entidades restantes (las unidades y los turnos) corresponden a cada uno de los ejes del Espacio-CSP.

Por lo tanto, podemos representar el problema en un espacio de dos dimensiones donde cada punto $P_{i,j}$ representa la i-ésima unidad disponible en el j-ésimo turno; y los candidatos son los conductores.

Supongamos que la empresa cuenta con 10 unidades. Hacemos corresponder al eje X las 10 unidades de la empresa, y al eje Y los tres turnos mencionados. De esta manera, el punto $P_{4,3}$ representa la cuarta unidad en el tercer turno (Noche).



El segmento Y=2 (G_1) abarca todas las unidades en el turno tarde. El Grupo 1 está definido por medio de los rangos $[1,l_1][2,2]$ (todos los puntos $P_{i,j}$ con j=2 e i $\in [1,l_1]$).



Una vez que el problema se encuadra como un Espacio-CSP, se pueden definir los grupos sobre los que se establecerán las restricciones y éstas corresponderán a algunos de los tipos de restricciones definidas. A continuación mencionamos un ejemplo para cada tipo de restricción:

Primer tipo: Los conductores 1 y 3 no pueden trabajar de noche.

Segundo tipo: Ningún conductor puede conducir más de una unidad en el mismo turno.

Tercer tipo: El conductor 5 debe entregar la unidad al conductor 7.

Cuarto tipo: El conductor 6 debe trabajar de mañana.

Quinto tipo: Los conductores que trabajan en un turno deben descansar en el siguiente.

6. CONCLUSIONES

- Se construye un modelo general que permite representar y resolver varios tipos de problemas de asignación de recursos en la industria. el comercio, etc., donde las restricciones son comunes a diferentes problemas y el espacio es el mismo.

- Las restricciones se organizan en clases de objetos y esto provoca reducción de redundancias; sin embargo, se pueden definir nuevas reglas y algoritmos de búsqueda específicos en el ambiente CP.

- Las distintas componentes del Espacio-CSP son representadas en el modelo orientado a objetos, dotándolas de todas las ventajas provistas por este paradigma, y por lo tanto, aumentando la declarativid.

- Tomando como punto de partida el modelo general propuesto para problemas de asignación de recursos, se pueden definir nuevas clases, o bien modificar las existentes para un problema particular siempre que este encuadre en el Espacio-CSP.

7. REFERENCIAS

[**Coad,91**] Object-Oriented Analysis. Peter Coad, Edward Yourdon.

[**Coad,91**] Object-Oriented Design. Peter Coad, Edward Yourdon.

[**Forradellas,95**] R.Forradellas, "Un Modelo para el tratamiento de Sistemas Dinámicos con Restricciones en el marco CLP", Tesis Doctoral, Universidad Politécnica de Madrid, España, 1995.

[**Gamma,94**] Design Patterns. Element of reusable object-oriented software. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

[**Ibañez,95**] F. Ibañez, "CLP(Temp), Integración de Restricciones Temporales Métricas y Simbólicas, en el Marco CLP", Tesis Doctoral, Universidad Politécnica de Valencia, España, 1994.

[**Ibañez,94**] F. Ibañez, R. Berlanga, F. Barber y R. Forradellas, "Dos enfoques de la Programación Lógica con Restricciones", Revista de Informática Teórica y Aplicada , Brasil, 4/1994.

[**Jaffar,94**] J. Jaffar and M.J. Maher, " Constraint Logic Programming: A Survey", J. Logic Programming, to appear, 1994.

[**Leler,90**] V. Leler, "Constraints Programming Languages - Their Specification and Generation", Addison-Wesley Publishing Co. Inc., 1990

[**Macworth,86**] A.K.Mackworth, "Constraint Satisfaction", Encyclopedia of Artificial Intelligence, 1986.

[**Martin,94**] Martin J. - Odell J., "Análisis y Diseño Orientado a Objetos", Prentice Hall Hispanoamérica S.A., 1994.

[**Puget,94**] Puget J., "A C++ Implementation of CLP", Ilog Solver Collected Papers, Ilog tech. report, 1994.

[**Rueda,95**] Rueda L. Klenzi R. Gutierrez L. Ibañez F. Forradellas R., "Tratamiento de Problemas de Combinatoria Discreta mediante el Paradigma CLP", 2das. Jornadas de Inteligencia Artificial, Universidad Nacional del Sur, Bahía Blanca, 1995.

[**Rueda,96**] L. Rueda, I.Arias, F.Ibañez and R.Forradellas, "Representación y Tratamiento mediante POO de Problemas de Combinatoria Discreta en la Programación con Restricciones", 25as. JAIIO, Argentina, 1996.

[**Rueda,97**] Rueda L. - Arias I. - Ibañez F. - Forradellas R., "Tecnologias de Inteligencia Artificial aplicadas a problemas Industriales Complejos ", VII RPIC, 1997.

[**Rumba,91**] Rumbaugh J - Blaha M. - Premerlani W. - Frederick E. - Lorensen W., "Object Oriented Modeling and Design", Prentice Hall, 1991.

[**Shlaer,88**] Object-Oriented Systems Analysis. Modeling the world in data. Sally Shlaer, Stephen J. Mellor.

[**Van Hentenryck,89**] P. Van Hentenryck, "Constraints Satisfaction in Logic Programming", MIT Press, 1989.

[**Van Hentenryck,92**] P. Van Hentenryck, H. Simonis and M. Dincbas "Constraint Satisfaction using CLP", Artif. Intell., Vol. 58, 1992.

[**Wirfs-Brock,90**] Designing Object-Oriented Software. Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener.

Cuarto tipo: El conductor 6 debe trabajar de mañana.

Quinto tipo: Los conductores que trabajan en un turno deben descansar en el siguiente.

6. CONCLUSIONES

- Se construye un modelo general que permite representar y resolver varios tipos de problemas de asignación de recursos en la industria, el comercio, etc., donde las restricciones son comunes a diferentes problemas y el espacio es el mismo.

- Las restricciones se organizan en clases de objetos y esto provoca reducción de redundancias; sin embargo, se pueden definir nuevas reglas y algoritmos de búsqueda específicos en el ambiente CP.

- Las distintas componentes del Espacio-CSP son representadas en el modelo orientado a objetos, dotándolas de todas las ventajas provistas por este paradigma, y por lo tanto, aumentando la declarativiidad.

- Tomando como punto de partida el modelo general propuesto para problemas de asignación de recursos, se pueden definir nuevas clases, o bien modificar las existentes para un problema particular siempre que este encuadre en el Espacio-CSP.

7. REFERENCIAS

[**Coad,91**] Object-Oriented Analysis. Peter Coad, Edward Yourdon.

[**Coad,91**] Object-Oriented Design. Peter Coad, Edward Yourdon.

[**Forradellas,95**] R.Forradellas, "Un Modelo para el tratamiento de Sistemas Dinámicos con Restricciones en el marco CLP", Tesis Doctoral, Universidad Politécnica de Madrid, España, 1995.

[**Gamma,94**] Design Patterns. Element of reusable object-oriented software. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

[**Ibañez,95**] F. Ibañez, "CLP(Temp), Integración de Restricciones Temporales Métricas y Simbólicas, en el Marco CLP", Tesis Doctoral, Universidad Politécnica de Valencia, España, 1994.

[**Ibañez,94**] F. Ibañez, R. Berlanga, F. Barber y R. Forradellas, "Dos enfoques de la Programación Lógica con Restricciones", Revista de Informática Teórica y Aplicada , Brasil, 4/1994.

[**Jaffar,94**] J. Jaffar and M.J. Maher, " Constraint Logic Programming: A Survey", J. Logic Programming, to appear, 1994.

[**Leler,90**] V. Leler, "Constraints Programming Languages - Their Specification and Generation", Addison-Wesley Publishing Co. Inc., 1990

[**Macworth,86**] A.K.Mackworth, "Constraint Satisfaction", Encyclopedia of Artificial Intelligence, 1986.

[**Martin,94**] Martin J. - Odell J., "Análisis y Diseño Orientado a Objetos", Prentice Hall Hispanoamérica S.A., 1994.

Aspect-oriented programming [KLM+97][AOP] is a new paradigm that intends to reduce the problem of code tangling, and therefore improve the quality of software. There are proposals of aspect oriented languages that present different ways in which aspects can be designed and programmed as separate and independent components of a system.

This paper presents a reflective architecture and an implementation that extends the Smalltalk environment in order to support aspect-oriented programming. This architecture supports the programming of cross-cutting aspects of a system in a separate and modular form, and composes these aspects with the corresponding functional modules at run-time, therefore avoiding the transformation of code and permitting the use of a general-purpose language such as Smalltalk both for the programming of the functional and aspect components of a system.

Aspect-Oriented Programming

Some features of software systems tend to affect or require the collaboration of groups of objects. They naturally *cross-cut* the primary division of functional modularity that is achieved with objects.

A design aspect is a feature that *cross-cuts* other components in the design. An implementation aspect is a program module consisting of variables, methods and weaves that captures the cross-cutting in a software system.

The goal of AOP is to provide methods and techniques for decomposing problems into a number of functional components (basic components) as well as a number of aspects which crosscut the functional components, and then composing these components and aspects to obtain system implementations.

The basic components are combined with the aspects by means of a *weaver*. This process can be carried out in a static way using a compiler or pre-processor by code transformation, as in AspectJ [AspectJ], or dynamically during run-time, using computational reflection.

The proposals of languages that support AOP mainly differ as to the manner in which the aspects and the composition (weaving) are programmed. These programming tasks may be carried out by means of an extension to an existing language or by a new and specific language.

Architecture and Implementation

The proposal presented in this paper consists of a reflective architecture for the incorporation of AOP to an object-oriented system and a prototype implementation in Smalltalk of this architecture.

Reflection is the capability of a computational system to reason about and act upon itself and adjust to changing conditions [Maes87]. The computational domain of a reflective system is the structure and the computations of the system itself.

In a reflective architecture, a computational system is viewed as incorporating an object part and a reflective part. The components of a reflective architecture reside at two levels: the base and the meta-level. The base level deals with the functionality of the application, and the meta-level deals with the application's self-representation. In an object oriented application, the objects that reside at the meta-level are called metaobjects.

The proposed architecture is built using the Luthier MOPs (Meta Object Protocols) framework that supports metaobjects, providing a flexible infrastructure for the association of metaobjects with classes, instances or specific methods [Campo97].

This framework for metaobject support provides a flexible and reusable reflection mechanism to the Smalltalk-80 system. The framework provides an infra-structure over which different policies of metaobject management can be built, based on managers and method interception models [Campo98].

The aspects are represented by objects, more precisely, by metaobjects. The Aspect class (which inherits from MetaObjectClass) is the generic class from which all aspect objects of the system inherit their behaviour.

The aspects are represented in a similar way to AspectJ [AspectJ]. The methods pertaining to an aspect are invoked either before or after the methods of the basic objects of the system, and as such are called before-methods or after-methods.

The weaver is represented by the AspectManager class, which provides the mechanism for the composition of the aspects by means of interception of messages.

When a message (at the base level) is intercepted, the control flow is redirected to the AspectManager class. The latter identifies the aspects (classes or instances) associated with the base object, and then does the following:

1. It executes all before-methods (Aspect's methods) associated with the message intercepted;
2. It executes the original message;
3. It executes all after-methods (Aspect's methods) associated with the message intercepted.

Finally, the normal control flow is restored.

The aspects are composed with the base objects by sending the following messages to an aspect: **composeBeforeMethod:with:** and **composeAfterMethod:with:**, where the parameters are the class and the methods to be intercepted.

This prototype permits an ample variety of crosscutting as shown below (Figure 1):

one aspect class	**crosscuts**	one class
many aspects classes		many classes
one aspect instance		one instance
many instances of different aspect classes		many instances of different classes

Figure 1: *Different kinds of crosscutting between classes and instances*

One of the main characteristics of this proposal are that the aspects and the components are both built in the same general purpose language (Smalltalk 80). Additionally, the use of a reflective architecture allows for the dynamic handling of the composition, permitting run-time modifications of the aspects without affecting the basic functional components of the system. As the functional components of the system are composed without modifications to the code of any of their classes or clients and no special "hooks" have to be provided, the composition is non-invasive. Also, as the aspects are programmed as separate classes and objects, the debugging problems associated with code transformation are avoided.

Conclusions

This paper has presented aspect-oriented programming as a new technique which provides solutions to an important deficiency of current component technologies: the inability to adequately capture cross-cutting aspects of software systems. An architecture and an implementation that supports aspect oriented programming has been developed as an extension to Smalltalk, which permits the incorporation of aspects to a system in a flexible and transparent way, though its dynamic handling of the weaving process may incur in certain runtime overhead.

References

[AOP] Homepage of Aspect-Oriented Programming Project Xerox Palo Alto Research Center (Xerox Parc), Palo Alto, California, http://www.parc.xerox.com/aop.

[Aspect J] Homepage of AspectJ™, Xerox Palo Alto Research Center (Xerox Parc), Palo Alto, California, http://www.parc.xerox.com/aop/aspectj.

[AOP-WR] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-Oriented Programming Workshop Report. ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä Finland, June 1997.

[Campo97] Marcelo Campo. "Compreensao Visual de Frameworks através da Introspecao de Exemplos". Ph.D. Thesis. UFRGS, Porto Alegre, Brasil., 1997. (in portuguese).

[Campo98] Marcelo Campo and Tom Price. "Luthier: : A Framework for Building Framework-Visualisation Tools" in "Object-Oriented Application Frameworks". Mohamed Fayad, Ralph Johnson (Eds.), John Wiley & Sons, USA, November 1998.

[KLM+97] G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J-M Loingtier, and J. Irwin. Aspect-Oriented Programming. In Proceedings ECOOP'97 – Object-Oriented Programming, 11th European Conference, Jyväskylä Finland, June 1997, Springer-Verlang, 1997.

[Maes87] Maes, P. "Concepts and Experiments in Computational Reflection". In Proceedings of OOPSLA '87.