Instituto de Computación – Facultad de Ingeniería
Universidad de la República Oriental del Uruguay

Anexo: Tesis de Maestría
en Ingeniería en Computación

Reuso de Reglas de Negocio: Una experiencia de reuso de ontologías en un dominio restringido.

Enrique Latorres

Supervisor: Juan Echagüe

Diciembre 2002

INDICE

Ejemplos del Ambiente Protégé-2000

En esta sección mostramos algunos de los desarrollos en ontologías y las herramientas desarrolladas para procesar las bases de conocimiento desarrolladas.

Instalación

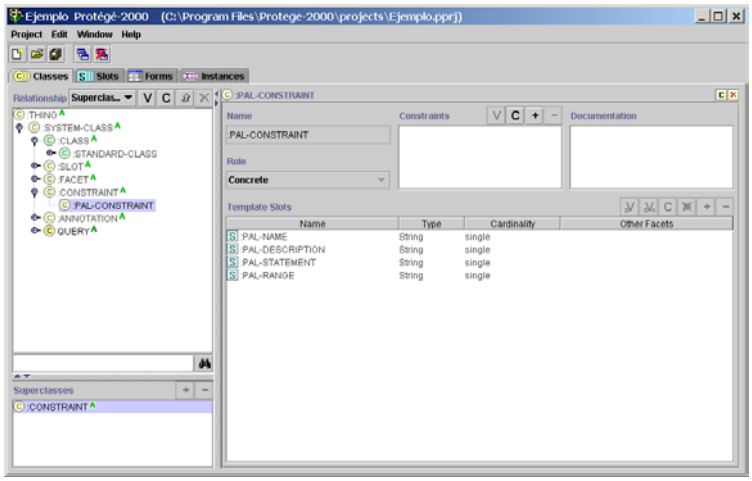
La instalación de Protege-2000 versión 1.7 no tiene dificultades. Sí es necesario que la máquina tenga un buen procesador, 800 Mhz o superior, y una buena cantidad de memoria, 256Mbyte o superior. Con cantidades inferiores puede haber comportamiento errático y retrasos en la respuesta considerables. Se recomienda hacer la instalación que incluye la máquina virtual Java para mayor facilidad de la instalación.

Antes de comenzar una nueva ontología asegurarse que el *plug-in* del PAL esté en el directorio de *plug-in's* y no olvidar hacer el *merge* del proyecto pal_query.pprj, si se va a hacer *queries* sobre la ontología. De lo contrario puede ser problemático crear *queries* y verificar que las restricciones son correctas, ya que no quedan incorporadas adecuadamente las Meta-Clases primitivas de las consultas. Para otros *plug-in* las meta-clases son incorporadas por el propio *plug-in* pero no en este caso.

Trabajando con la herramienta

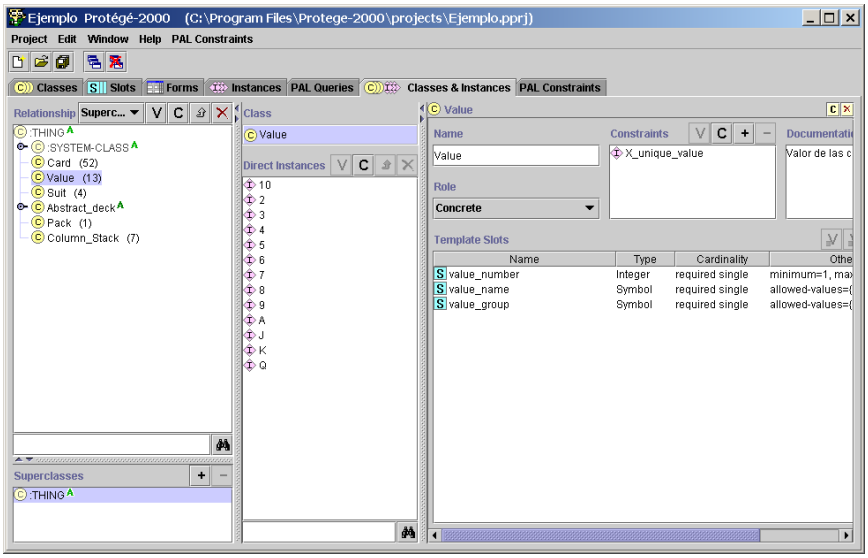
Si se van a usar consultas y restricciones de PAL, no se puede usar un proyecto nuevo, vacío, sin las meta clases de PAL. Por lo tanto se debe abrir el proyecto pal_query.pprj y salvarlo con el nuevo nombre, o realizar un *merge* de este en el nuevo proyecto. De lo contrario las consultas no se van a presentar en forma adecuada en los formularios para editar y consultar.

Al revisar la base de conocimiento generada vemos que dispone de las meta-clases correspondientes al manejador de consultas y restricciones PAL.

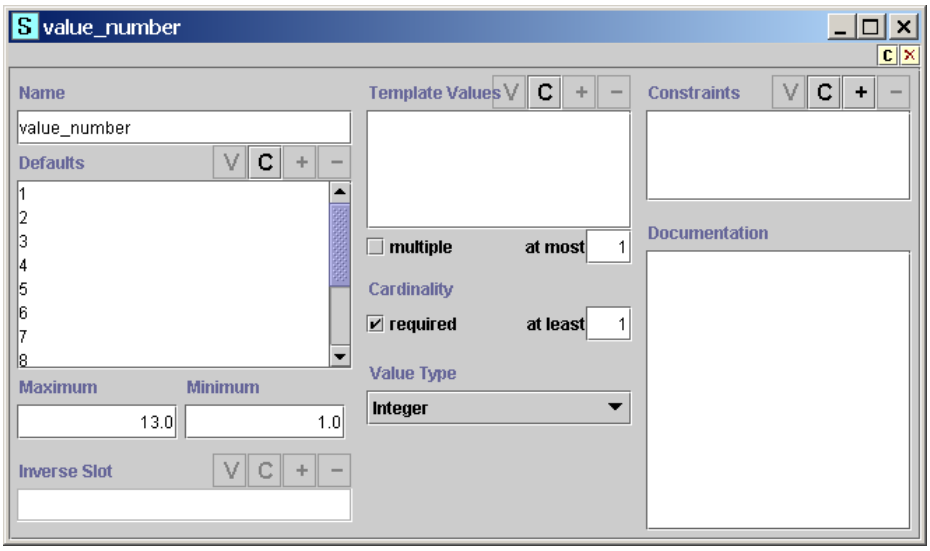


Un *plug-in* muy práctico es el que permite editar clases e instancias sobre un misma vista. Y se puede navegar en los diferentes niveles con un clic. Su instalación es opcional. En general la instalación de un *plug-in* consiste simplemente dejar el archivo ".jar" asociado en la carpeta de *plugins* de Protégé-2000 y reiniciarlo.

En este caso se ve la clase Value con sus instancias que van a dar valor a las cartas. Los slot de esta clase consisten en un número, un nombre y un grupo.

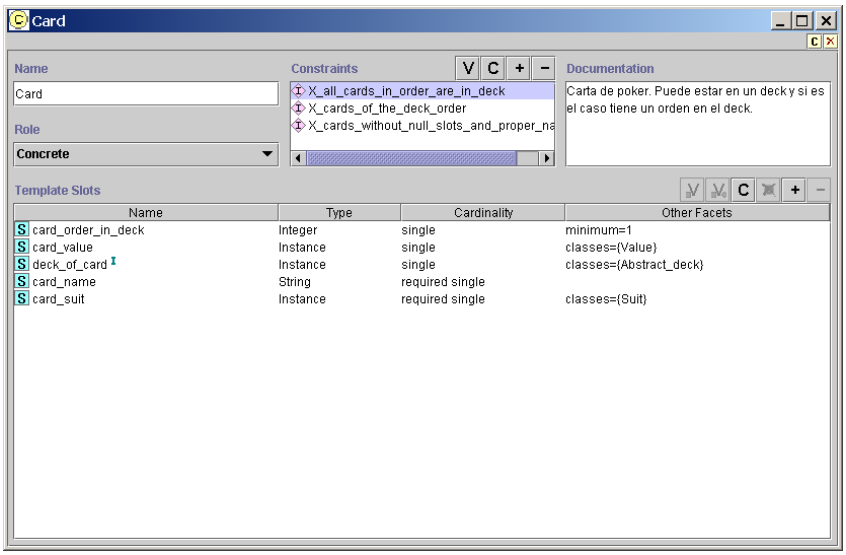


Si se hace doble clic sobre un slot se puede ver su definición. En ella se ingresan algunas de las restricciones básicas para la estructura, como ser rango de valores, multiplicidad, clases a las que hace referencia, incluso si existe un slot inverso en otra clase que deba ser mantenido en coordinación con este, como ser un slot "padre de" debe coincidir inversamente con un slot "hijo de" para las instancias correspondientes que mantienen esa relación.

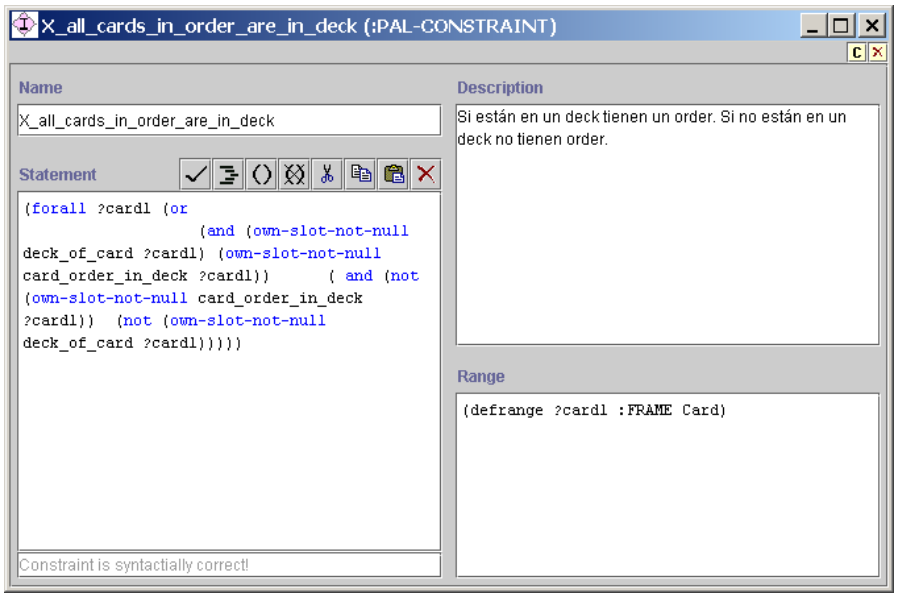


Para cada clase o slot se puede asociar restricciones de forma que las clases o slot asociados estén declaradas en forma automática como rangos posibles de instancias sobre las que se apliquen.

En este caso mostramos la clase carta con las restricciones asociadas en el campo correspondiente.



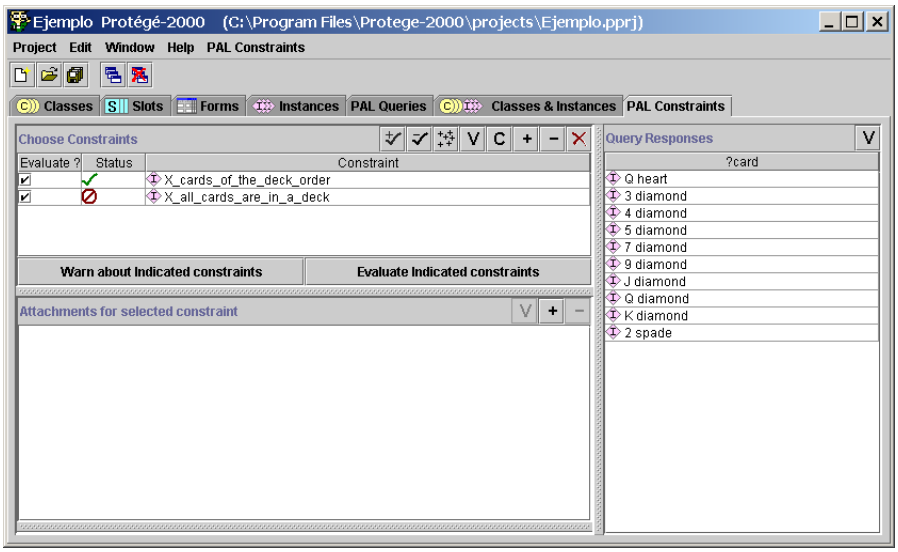
Si seleccionamos una restricción para analizarla o editarla nos aparece el siguiente formulario:



Esta restricción exige que si una carta está en un montón debe tener un orden asociado, o sea que debe ser la carta numero X del montón pero el slot no puede ser nulo. Otra restricción asegura que todas las cartas en un montón estén en orden a partir de una primera. Haciendo clic en el boton de checkmark, el editor hace una evaluación sintáctica de la declaración y puede indicar alguna sugerencia para corregirlo si hay un problema. Muchas veces

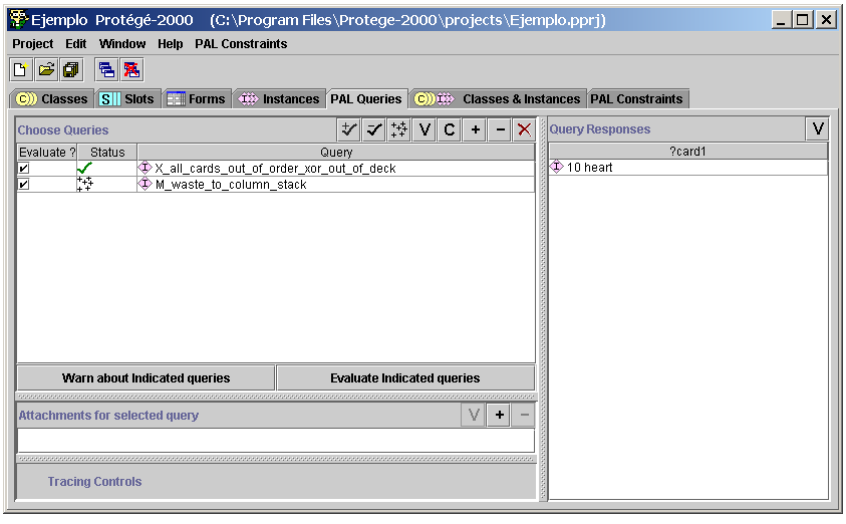
simplemente no informa nada. Si informa “Constraint is sintactically correct!” podemos seguir con lo siguiente, que es la prueba de la restricción.

Se puede ir al formulario de restricciones y elegir las que se van a evaluar. En particular en este ejemplo se toman dos restricciones y se evalúan, una dio que no hay instancias que infrinjan esa restricción la otra en cambio que verifica que no haya cartas fuera de un montón, encontró 10 cartas que no la verifican.

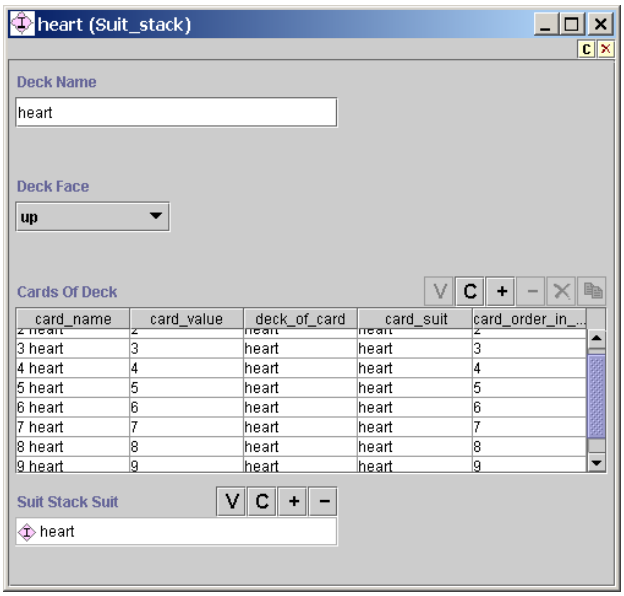


Para corregir los experimentos se debió desarrollar un conjunto de consultas dentro de la herramienta, que permitiera identificar instancias que infringen las restricciones. Las consultas y restricciones de elementos que inflingen reglas se nombrarán con *X_nombre_de_la_consulta*. Mientras que las restricciones y consultas relativas a jugadas posibles son denotadas *M_nombre_de_la_consulta*. Además se desarrolló un conjunto de programas para post-procesar las bases de conocimiento y obtener las métricas de reutilización.

En el formulario de consultas podemos elegir las que deseemos evaluar y obtenemos la lista de instancias que verifican la consulta.



Podemos también trabajar con las instancias y asociar valores a los *slots*, en particular este caso presenta un montón de cartas de un palo y la lista de instancias de cartas asociadas.



De esta forma se presentó una rápida introducción al ambiente de trabajo en la herramienta Protégé-2000 y el aspecto de su interfase.

Justificación de un paradigma objetivo

Planteamos aquí un conjunto de ideas que están dirigiendo esta línea de investigación.

Consideremos un sistema ideal para el desarrollo de sistemas empresariales, de un dominio de conocimiento o de negocio, nuestro sistema debe ser flexible y útil para el apoyo a la ingeniería de requisitos. Podría ser un sistema basado en conocimiento, usando técnicas de reconocimiento y clasificación de patrones de conocimiento, que asista al ingeniero en requisitos/conocimiento/consultor en el relevamiento de las reglas que rigen un cierto dominio de negocio.

El consultor al ingresar una regla, que se obtiene a partir de entrevistas con el usuario, obtiene una respuesta del sistema. Esta respuesta puede ser, entre otras, una sugerencia sobre nuevas preguntas relativas a reglas relacionadas al tema o la regla que se acaba de ingresar. El sistema acumuló información en las experiencias con otras organizaciones que indican el vínculo entre reglas similares a la recién ingresada y otras en la base de reglas del sistema.

El sistema podría disponer de la capacidad de generalizar reglas que fueron aplicadas a dominios de negocio de diferente giro, pero que dadas las propiedades identificadas, el sistema es capaz de resolver problemas por similitud identificando patrones, con un estilo similar a **[Stor1997]** donde se

identifican patrones estructurales de bases de datos, y aplicando técnicas similares al Razonamiento Basado en Casos **[Broa2000]**. Esto resolvería algunos de los problemas fundamentales descubiertos en **[Coh1999]** como ser la divergencia entre ontologías desarrolladas por grupos diferentes cuando no disponen de un repositorio común de donde extraer conceptos reutilizables.

El sistema podría disponer de definiciones que se realizan a nivel conceptual, el más alto nivel de abstracción posible, y se vinculan a múltiples versiones de mas bajo nivel de abstracción con cada concepto que serán reutilizables y reemplazables en forma lo más automática posible. El ambiente de desarrollo podría enlazar conceptos con vinculaciones flojas, tanto que el pase de parámetros no sea obligatorio sino a demanda del concepto que lo necesite, por la necesidad de saber y para poder hacer, al estilo de un agente que interactúa con otros para tratar de conseguir los recursos necesarios para cumplir su función **[Woo2000]**.

Al final una vez ingresados los requisitos como restricciones y definiciones dentro del sistema y habiendo reutilizado todo el conocimiento posible de las experiencias pasadas con el sistema, la generación del código puede ser automática o guiada por temas estéticos o de requisitos del sistema para producir el ejecutable que implementará los procesos registrados en la base.

El actual ingeniero de software se convertirá en un ingeniero de conocimiento o de requisitos y sus habilidades se vincularán más a técnicas lógicas, psicológicas, sociológicas y archivológicas para escudriñar y operar con el conocimiento explícito e implícito de una organización.

Esos mismos registros del conocimiento de la organización podrán ser usados para definir manuales de procedimiento, para entrenar nuevos empleados y que conozcan la realidad corporativa de la organización, para consultas en línea sobre reglas de la organización, mediante los sistemas y "traductores" adecuados, y así lograr la formalización, publicación y reutilización del conocimiento de la organización.

¿Ciencia ficción? Aún sí, pues para que esto funcione deben resolverse varios problemas. Pero también hay mucha gente tratando de atacarlos. Uno de los principales problemas es definir mecanismos adecuados para poder garantizar la integración de conocimiento de distintas fuentes en forma coherente y tratable, dentro de una base de conocimiento. Hay múltiples propuestas pero las que mejor se definen tienen aún dependencia de la implementación, por lo tanto están lejos de la reutilización conceptual.

Antecedentes de la investigación

En este experimento se busca obtener un conjunto de métricas en forma automática, sobre el estudio y comparación de varias ontologías sobre dominios similares, desarrollados con criterios similares. Se espera poder definir antecedentes para esta línea de investigación sobre uso de ontologías para el desarrollo de sistemas, que incluye especificación formal de sistemas, especificación conceptual de los mismos, reuso e integración de

especificaciones y generación automática de código, manuales e instructivos para formación a partir de estas especificaciones, basadas en ontologías.

Otras referencias a métricas de reuso en sistemas de gestión de conocimiento son **[Con2000] [Usc1998]** y **[Coh1999]**. El primero analiza varias implementaciones de una Conceptualización común presentada a varios equipos de trabajo. Se realizan varios equipos con estudiantes de diversa experiencia en la modelización de ontologías, y se definen métricas con similares a las de este documento pero solo se verifica la taxonomía de las clases (*frames-slots*), al parecer sin contar las reglas de restricciones. Definen una métrica derivada de la métrica de distancia para cadenas de Levenshtein **[Lev1966]**. El problema fácilmente se puede vincular a la comparación de árboles sintácticos aquí presentado. El segundo trabajo **[Usc1998]** se restringe a la aplicación de una única ontología en un programa de ingeniería, como reuso de un sistema de resolución de problemas, y el último [Coh1999] al reuso de ontologías para un sistema experto que responda a un conjunto de “competency questions”.

Dado un dominio de conocimiento determinado y dos implementaciones particulares dentro de ese dominio, si se analizan las reglas que definen las actividades de uno y otro a nivel conceptual debe haber una gran cantidad de conocimiento común y de reglas que son fundamentalmente similares.

Ya sea este un dominio de negocio, con su implementación particular de una empresa y sus procesos de negocio comparado a otro del mismo tipo, o un ámbito científico, dinámico y cambiante, incluso un laboratorio en particular con sus creencias particulares, comparado a otro, etc. Siempre deben compartir un conjunto importante de reglas que definen su forma de llevar adelante las tareas, deducciones y procesos. Y existe un conjunto relativamente pequeño de reglas que definirán la idiosincrasia particular de cada organismo. Y quizás esas reglas de idiosincrasia puedan ser encapsuladas como versiones o implementaciones particulares de modelos o conceptualizaciones superiores.

En particular esa información debería ser expresada a nivel conceptual, donde la experiencia muestra que esa información conceptual al menos mediante los mecanismos informales de boca en boca, entrenamiento a otras personas, y otras formas humanas de comunicación, sirven y funcionan desde los primeros tiempos de la humanidad. Sin un cierto nivel compartido de conceptos y criterios sería imposible que las organizaciones funcionaran.

Hoy sabemos que el factor humano afecta el desempeño y la predecibilidad de las tareas y funciones de cualquier organización. Así también el factor humano afecta en que no es rápido para adaptarse a los cambios que exigen mucho dominios de negocios. Precisan mucha capacitación y entrenamiento para que puedan realizar en forma eficiente nuevas tareas, con más tiempo de entrenamiento cuanto más compleja sea la tarea.

Rápidamente, la gente de consultoría y las organizaciones altamente dinámicas se percataron que, para acelerar la velocidad con que las

personas, y por tanto las Organizaciones, se adaptan a nuevas condiciones de trabajo y nuevas formas de llevar adelante las tareas, era necesario hacer que estas tuvieran un soporte informático potente y flexible. En la flexibilidad para adaptarse a los nuevos cambios del software, iba apoyada la factibilidad que la organización pudiera hacer lo mismo con las nuevas condiciones del mercado.

Bajo esta hipótesis entonces buena parte del problema es asegurarnos que la organización pueda desarrollar las modificaciones a los sistemas informáticos que el mercado y las nuevas condiciones exigen. Pero esto no es tan fácil. La "crisis del software" que ya lleva más de treinta años no parece tener una solución efectiva **[Broo1987]**. Brooks (1987) reconoce que "...la problemática está más en la especificación, diseño y construcción de las construcciones conceptuales que en la representación y la fidelidad de esa representación. Aún hacemos errores de sintaxis, es seguro; pero son trivialidades comparado con los errores conceptuales en muchos sistemas". Problemas similares son reconocidos por William Clancey en **[Clan1993]**.

Brooks (op. cit.) enumera las propiedades de lo que el define como esencia irreductible de los modernos sistemas de software: La complejidad, la concordancia[1], la cualidad de cambiable[2] y su invisibilidad. Para cada uno de esto atributos alega la razón por la cual son prácticamente intratables, y muestra como muchos de los avances en las técnicas y herramientas de Ingeniería de Software, solo han resuelto un conjunto de dificultades accidentales pero no han atacado estas cuatro raíces del problema.

Una crítica a Brooks es contestada ocho años después por Brad J. Cox asegurando que la "Bala de Plata" sí existe, pero hay que forjarla y que el mayor problema es que exige un cambio de paradigma que va a impactar en las más profundas convicciones de aquellos a quienes incumbe, y que se van a resistir **[Cox1995]**.

Una de las propuestas es que la "complejidad encapsulada" deja de ser complejidad, en la medida que los bienes se usan libremente sin necesidad de manejar los antecedentes de cómo fueron construidos, o cuantas personas estuvieron involucradas. Obviamente para muchos objetos físicos es elemental que cualquier persona sepa como usarlos, como por ejemplo un lápiz, sin importar la complejidad de cómo fueron creados y puestos a disposición.

Esa complejidad de su proceso de elaboración, está eficientemente oculta en las sucesivas etapas de su elaboración en un árbol-cadena de proveedores y clientes. El software en cambio no es desarrollado con el nivel de granularidad en sus componentes de igual forma que los productos físicos. Cox consolida otros conceptos propios anteriores como los componentes comercializables **[Cox1986]** con la idea de pago por uso, como propuesta para

[1] Del Inglés "Conformity", cumplir con los requisitos.
[2] Del Inglés "Changeability", que sufre o está sujeto a cambios.

elaborar un mercado de componentes reusables que tenga similitudes con el mercado de los bienes de átomos y moléculas.

Creo que la propuesta no es exclusiva de Cox, pero es muy pronto para poder presentar esta forma de llevar adelante la comercialización del software y no hay una infraestructura de comunicaciones adecuada para soportarlo. De todas maneras no resuelve otros problemas como ser la variabilidad de plataformas y condiciones de ambiente para facilitar el reuso de componentes.

Como resumen los problemas fundamentales de las ingeniería del software se llevan hacia el lado de la interfase humana, y es la razón humana la principal causa de estas dificultades. Una de las principales fuentes es la diversidad de concepciones aún en conceptos comunes, y triviales. Estas variaciones en concepción causan problemas de comprensión al especificar sobre un dominio en común. Para esto viene en nuestra ayuda el uso de ontologías y técnicas de gestión del conocimiento.

Pero si se va a aplicar un cambio de paradigma en algo tan importante como las técnicas de ingeniería de software, esto solo será posible si el conjunto de beneficios directos es mayor que los esfuerzos necesarios.

El escenario que se plantea en la sección anterior es un objetivo ideal, no practicable actualmente, que afecte la problemática fundamental de la Ingeniería del Software, para transformarla en Ingeniería del Conocimiento. Esto sería posible, de resolverse y automatizarse todos los procesos intermedios que se vinculan con la construcción del software.

Una problemática fundamental que aún no está resuelta es la de la integración automática o semiautomática de conocimiento de orígenes múltiples para la reutilización y construcción de nuevo conocimiento.

Una propuesta en su mayor parte automática se puede encontrar en **[Ome22001]** donde describe un algoritmo de integración (*merge*) de ontologías. Este se basa en diccionario de sinónimos, una función de distancia entre los mismos y que sugiere un conjunto de operaciones para aplicar en las definiciones de clases (*frames*). El diccionario de sinónimos es usado a partir de WordNet **[Fell1998]**, una Ontología de Lenguaje Natural. Si bien el sistema sugerido tiene un enfoque de integración sintáctica, que aplica reglas en función de los nombres de las clases, atributos y relaciones, no garantiza la tratabilidad (*tractable*) de las reglas generadas, ni que conceptualmente el resultado tenga significado, debe ser verificado por un humano.

Probablemente no podamos nunca evitar que algunas de las reglas generadas deban en algún momento ser verificadas por un operador humano. ¿Pero si esas reglas comienzan a ser excesivamente complejas, lo podrá realizar en forma efectiva?

Otra propuesta es la de **[Pint2001]** de la que hay varias versiones de esta propuesta pues fue evolucionando en el tiempo. Básicamente se apoya en

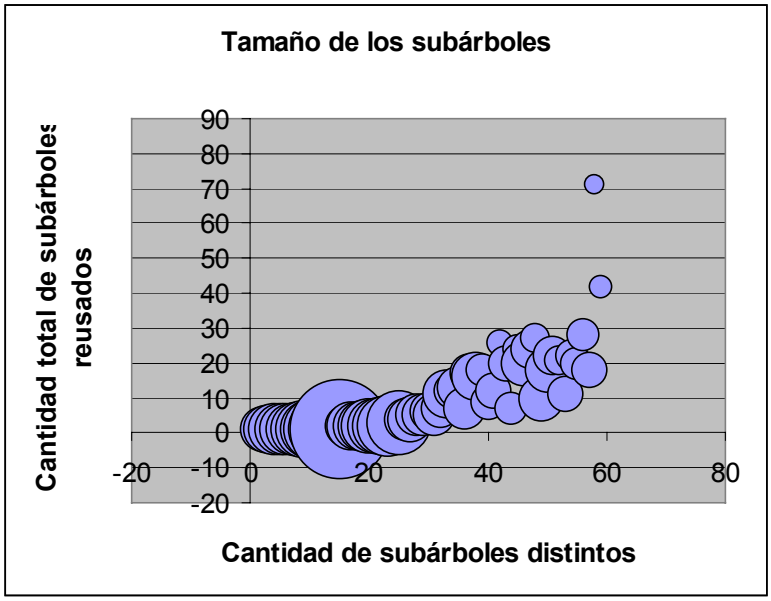
trabajos anteriores, pero define una metodología, un proceso para la integración de ontologías. Reclama en uno de sus párrafos la dificultad de conseguir ontologías con un adecuado nivel de granularidad conceptual. Esto es como consecuencia de que pocos de los trabajos están orientados a un ciclo completo del uso de conocimiento, y muchos solamente a la investigación de algoritmos y procedimientos de manejo de ontologías, por lo que procesan ontologías que no contienen suficiente especificidad de un dominio de conocimiento. Aún así para evaluar las factibilidad de integración [Pint2001] se basa aún en parámetros difíciles de objetivar totalmente.

Es evidente que hace falta aún trabajo en estas áreas.

Lo que se plantea entonces es apenas un paso en la búsqueda de técnicas para la integración automática de conocimiento. En particular, la factibilidad de aplicar reconocedores de patrones a las especificaciones, la comparación de reglas por similaridad y el reuso de reglas dentro de un determinado dominio de conocimiento.

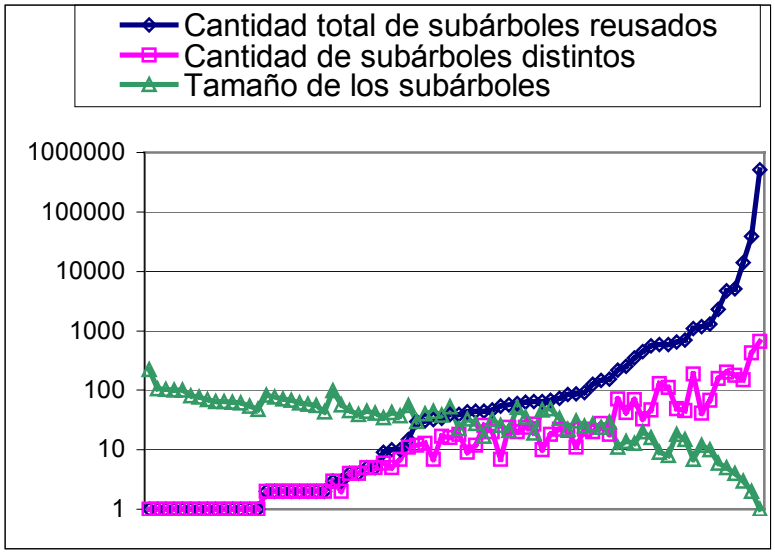
Elementos sobre la distribución de subárboles interesantes.

Comparando la Cantidad total de subárboles reusados contra la Cantidad de subárboles distintos para un tamaño dado de subárboles vemos que hasta el de 30 reusos y 60 subárboles mantienen una cierta correlación entre el primero y segundo parámetro.

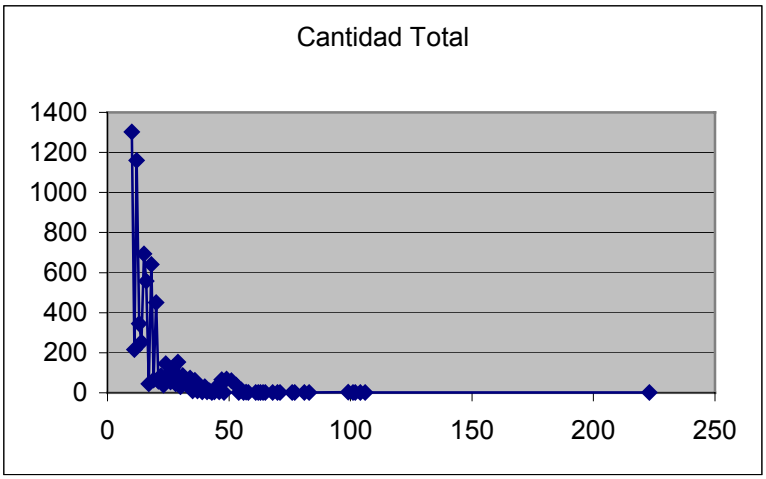


Como vemos en el cuadro anterior, a medida que la Cantidad total de subárboles reusados crece la Cantidad de subárboles distintos no aumenta en la misma proporción. La variabilidad de los árboles disminuye notablemente con relación a la cantidad de subárboles reutilizados y también disminuye el tamaño a menos de 20 tokens.

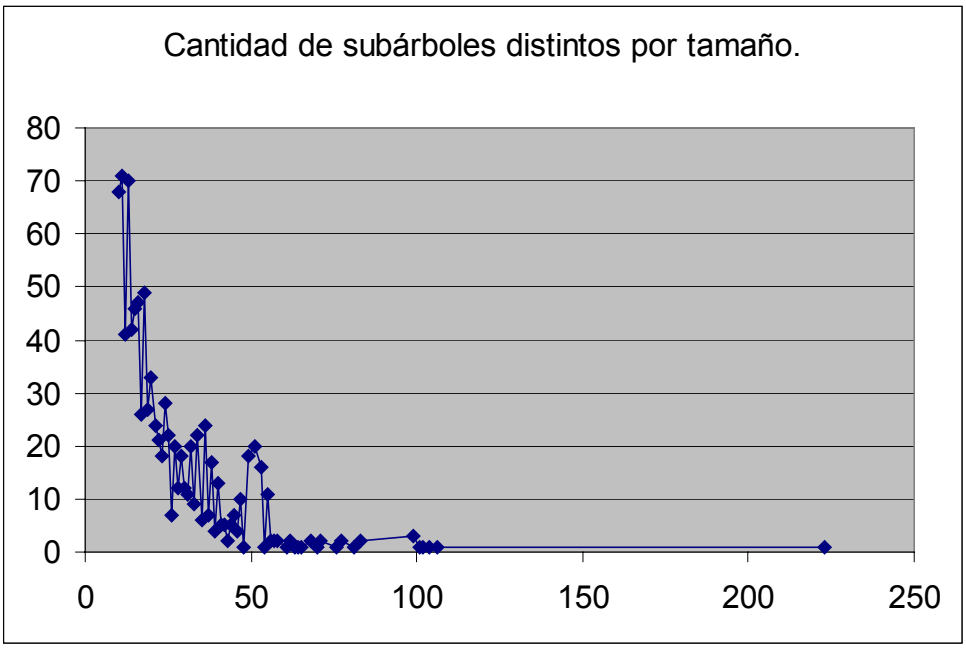
Puestos en una escala logarítmica ordenados por Cantidad total de subárboles reusados se notan tres áreas más o menos definidas donde las características fundamentales de los árboles reutilizados es bastante diferente. Una primer área donde la Cantidad total de subárboles reusados y la Cantidad de subárboles distintos se mantienen casi lineales uno al otro, con valores bajos, y el tamaño de los subárboles es el mayor posible. En esta área se ve la reutilización de grandes conjuntos de componentes, clases enteras, restricciones completas, conjuntos de instancias, etc. Luego hay un área de Subárboles intermedia donde hay poca variación de tamaño y de Cantidad de subárboles diferentes a medida que aumenta la cantidad de árboles reusados. En esa área notamos que los picos locales de tamaño coinciden con los conjuntos de subárboles que se identificaron anteriormente. Si el cuadro anterior se ordena por tamaño de los subárboles obtenemos una gráfica similar a la vista previamente.



los subárboles interesantes como mayores a 10 tokens y quitamos el subárbol máximo de comparación. Con este conjunto a partir del conjunto de todos los subárboles reutilizados tenemos la siguiente distribución:



La zona más interesante para analizar posibles subárboles a reutilizar es entre los tamaños 43 y 54, con cantidades de reutilización que van desde 1 hasta 65.



Al analizar los subárboles incluidos dentro de ese rango vemos que varios corresponden a conceptos reutilizables, aunque dadas las dimensiones modestas del experimento ya habían sido notados en el proceso de desarrollo.

Una posibilidad es que al igual que ciertos ambientes de programación tenga facilidades de “autotexto” o sea que a medida que el desarrollador escribe reglas le sugiera conjuntos más o menos grandes de reglas para incorporar de una sola tecla. Pero esto no necesariamente ayuda a la comprensión y el encapsulamiento de conocimiento. Se debe agregar etapas de clasificación y asignación de semántica a esas reglas para que sean útiles y fácilmente reutilizables.

De los experimentos y temas de planificación.

Para hacer un análisis más preciso se debería analizar cada nodo y sus hijos en función del operador. Esto significa que un análisis similar al definido antes, que llevara la cuenta del significado semántico de algunos *tokens* podría devolver coeficientes de similaridad más acertados.

Al menos en teoría se debe esperar que si se usa como repositorio de reglas a reutilizar un conjunto de reglas más amplio que incluya todas las reglas de otros modelos que se usaron o se podrían haber usado, el nivel de reutilización debería ser más alto que comparado con cualquier ontología individual. Sin embargo, de nuestra experiencia no se obtiene que disponer

de muchas bases de conocimiento integradas sea mucho mejor a elegir la mejor de todas. La ventaja es que tendremos una o varias mejores si contamos con la herramienta, mientras que puede que no sea posible si solo se dispone de un conjunto limitado de bases de conocimiento.

El disponer de un conjunto de subárboles con niveles de reutilización reconocidos puede ayudar al sistema a recomendar al desarrollador algunos subárboles a medida que se van ingresando y se reconocen los patrones de estructura. Se pueden determinar algunos criterios para filtrar los subárboles más interesantes de ser reutilizados en base a definiciones conceptuales asignándoles una semántica, pero ese trabajo depende de un operador humano. Claramente esto resolvería varios de los problemas planteados en [Coh1999] en particular los de la divergencia de las ontologías debido a la dificultad de encontrar términos y subconjuntos reutilizables. En nuestro caso planteamos el reuso de componentes aún más pequeños que el de términos en un “Upper Ontology” (Ontología de términos básicos).

El criterio de similaridad se debe extender a continuar un cierto número de nodos luego del nodo diferente y ver si el subárbol vuelve a resincronizarse con el subárbol patrón. Esto vemos que se asocia a agregar una restricción a una de las instancias consideradas, por ejemplo la regla que determina la elección de una carta cualquiera de un mazo, con otra que elige la última. Esta forma de medir los subárboles seguramente dé niveles más certeros de la reutilización que mejoren las cifras aquí presentadas.

Hay trabajo para seguir haciendo. Por ejemplo un conjunto de extensiones a Protege-2000 que implementen algunas de las ideas aquí planteadas. Hay que probar si estos niveles de reuso se cumplen para la descripción de reglas de negocio, u otros dominios. Hay que plantear extensiones al lenguaje para incorporar operaciones con las instancias, generadoras de nuevas reglas, y eventualmente traductores a código ejecutable que creen una simulación de la realidad planteada. En particular mecanismos que permitan codificar reglas constructivas de la información de la base de conocimiento. Hay que incorporar al lenguaje de algunas primitivas de tipos abstractos de datos de forma de evitar construcciones artificiosas para representar esas estructuras.

En los hechos quedaron planteadas algunas interrogantes además de las respuestas que se obtuvieron, pero definen un camino de trabajos a realizar en este sentido.

Particularidades y Experiencias del ambiente de trabajo

Problemas encontrados

El editor de *statements* tiene algunos *bugs* que hacen que se pierda texto o no se pueda agregar o no se vea texto que está en el *slot*. No conviene hacer líneas muy largas sin un cambio de línea pues empieza a tener errores que en la consola del motor java los indica como de punteros. Ahora si se abusa

con los cambios de línea empieza a no poder procesar las últimas declaraciones incluso pueden que desaparezcan si se solicita que las indente.

La herramienta muestra que no fue ideada en principio para trabajar con restricciones complejas. Ya con seis instancias comienza a advertir que la consulta es muy compleja pero no permite encadenar o componer restricciones en base a otras más simples.

```
M_column_stack_to_foundation, Es mover la carta que está en el tope del alternating
color stack al Suit Stack correspondiente. No considera el caso que el suit stack esté
vacío. Se maneja en M_column_stack_to_empty_foundation

  Sentence may be overly complex (it uses 6 variables).

  (FRAME Card cards_of_deck)
  (defrange ?card2 :FRAME Card cards_of_deck)
  (defrange ?card3 :FRAME Card cards_of_deck)
  (defrange ?card4 :FRAME Card cards_of_deck)
  (defrange ?altc_stack :FRAME Alternating_color_stack)
  (defrange ?suit_stack :FRAME Suit_stack)
  (exists ?card1
    (exists ?altc_stack
      (and
        (and
          (deck_of_card ?card1 ?altc_stack)
          (forall ?card2
            (=>
              (and
                (deck_of_card ?card2 ?altc_stack)
                (/= ?card1 ?card2))
              (>
                (coerce-to-integer
                  (card_order_in_deck ?card1))
                (coerce-to-integer
                  (card_order_in_deck ?card2))))))
        (exists ?suit_stack
          (exists ?card3
            (and
              (and
                (deck_of_card ?card3 ?suit_stack)
                (forall ?card4
                  (=>
                    (and
                      (deck_of_card ?card2 ?suit_stack)
                      (/= ?card3 ?card4))
                    (>
                      (coerce-to-integer
                        (card_order_in_deck ?card3))
                      (coerce-to-integer
                        (card_order_in_deck ?card4))))))
              (and
                (=
                  (value_number
                    (card_value ?card1))
                  (+
                    (value_number
                      (card_value ?card3))1))
                (=
                  (card_suit ?card1)
                  (card_suit ?card3)))))))))
```

Ha sucedido que debido a algo que quedó en el código de las restricciones da errores cuando se trata de evaluar las restricciones y no llega a hacerlo, pero muestra mensajes de error en la consola. Algunas veces se ha resuelto borrando toda la restricción y haciéndola de nuevo y otras entrando directamente en el archivo .pins de instancias y editando las sentencias allí.

El analizador sintáctico se confunde a veces pues no delimita los tokens con espacios. Por ejemplo se estaba evaluando si una carta era la primera del deck (order_in_deck ¿card 1) pero lo interpretaba como or seguido de "der_in_deck", por lo tanto fallaba. Se optó por definir los nombres de slots que no coincidan al comienzo con palabras reservadas de PAL-KIF.

Hay veces que al crearse una nueva clase del sistema por un plug-in no se refresca correctamente la vista de clases y no se puede ver la clase por tanto tampoco las instancias. Hay que salir y volver a entrar (viejo truco).

El siguiente problema es la interpretación que hace el sistema del lenguaje. Uno asume que para una instancia determinada de tipo Card la sentencia (card_order_in_deck ?card 1) se verificara para todas las cartas que son la primera. Pero al hacer una consulta con esta sentencia no responde ninguna carta. En particular la consulta siguiente no responde ninguna instancia:

```
(defrange ?card :FRAME Card)
(findall ?card
  (and
    (own-slot-not-null card_order_in_deck ?card)
    (card_order_in_deck ?card 1)
  )
)
```

Hay que ver otra forma de representar la consulta que sea satisfacible por el motor de consultas. Hubo de cambiarse a la siguiente, pues la comparación de un slot de tipo entero con un número no funcionaba. Se debió explicitar la coerción de tipos.

```
(findall ?card
 ( and
  (own-slot-not-null card_order_in_deck ?card)
  (= (coerce-to-integer (card_order_in_deck ?card)) 1))
```

Es interesante notar que en todos los ejemplos de la bibliografía, la otra notación es la más presentada, y que eventualmente funciona. Pero para estar seguro conviene evaluar una a una todas las sentencias.

El parser-intérprete del lenguaje no está bien pulido. Puede ser necesario que se deban hacer varias versiones de la restricción y deben ser testeadas para verificar que cumplen con lo esperado. El manejo de tipos es, por lo menos, incierto. Y las sentencias como ser = o /= no es claro que funcionen en todos los casos. Hay casos donde se ve que (= (slot1 ?instance1) (slot1 ?instance2)) no siempre se evalúa igual que (slot1 ?instance1 (slot1 ?instance2)). Esta última es preferible si es posible.

El analizador sintáctico puede indicar que está correcta la sintaxis pero se está haciendo referencia a términos no definidos en el rango y no lo detecta. Por lo tanto si la consulta o la restricción no cumple con lo que se espera hay

que revisar uno a uno los términos a ver si están bien escritos, y si fueron declarados en el rango.

En muchos casos de restricciones largas conviene ir probando la cláusula en versiones más cortas y poner stubs en las restricciones que aún no se han agregado, para asegurarse que el motor de consultas entienda lo que uno pretende. Cuando el subconjunto de la consulta hace lo que se espera se agregan más restricciones para completarla.

Al definir las clases conviene hacer uso de la herencia y diferenciar roles aún cuando su implementación sea idéntica pues en las restricciones es difícil asociarlas a instancias (están pensadas para clases y estructuras). Es más sencillo definir una sub-clase con el nombre del rol y luego la instancia única (si ese fuera el caso) que cumple ese rol.

Para el siguiente ejemplo vemos que hay que cuidar el anidado de las cláusulas, los casos para los que son verdaderos o falsos. En particular la cláusula de implicancia se evalúa verdadero si para cada vez que la primer cláusula es verdadera la segunda también lo es, pero si no se cumple la primer cláusula el predicado de implicancia es verdadero, entonces si no hay ninguna carta en el waste simplemente no va a completarse el exists ?card1 con sus restricciones, pero si hay solo una carta no se va a encontrar otra ?card2 que verifique la primér cláusula de la implicancia y por lo tanto va a dar verdadero para todos los casos. Entonces el caso siguiente:

```
(defrange ?card1 :FRAME Card cards_of_deck)
(defrange ?card2 :FRAME Card cards_of_deck)
(defrange ?waste :FRAME Waste)

 (exists ?card1
 (exists ?waste
  (and
   (and
    (deck_of_card ?card1 ?waste)
    (or /* NO SIRVE PARA NADA; NO AGREGA NADA */
     (= (number-of-slot-values cards_of_deck ?waste) 1)
     (forall ?card2
      (=> (and (deck_of_card ?card2 ?waste)
               (/= ?card1 ?card2))
          (> (card_order_in_deck ?card1)
             (card_order_in_deck ?card2))
      )
     )
    )
   )
   (Predicate))))
```

La siguiente cláusula tiene el mismo resultado.

```
(exists ?card1
 (exists ?waste
  (and
   (and
    (deck_of_card ?card1 ?waste)
    (forall ?card2
     (=> (and (deck_of_card ?card2 ?waste)
              (/= ?card1 ?card2))
         (> (card_order_in_deck ?card1)
            (card_order_in_deck ?card2))
     )
    )
   )
   (Predicate))))
```

Las dificultades con el editor y el analizador sintáctico obligaron a que se desarrollara un editor externo (IndentEd) con un indentador parametrizable que permite indentar y analizar los niveles de paréntesis para asegurar la sintaxis de la restricción. Luego la restricción se lleva a un formato de indentación que minimiza los cambios de línea y se copia y pega en Protégé-2000.

Sugerencias y mejoras al ambiente

Patrones

La siguiente declaración resulta como ejemplo de patrón para definir una carta que pertenece a un deck, un waste (montón de cartas con cara vista) en este caso y es la disponible a utilizar, y en el lugar del `(Predicate)` se pueden poner las restricciones extra que esa carta debe cumplir.

```
(exists ?card1
 (exists ?waste
  (and
   (and
    (deck_of_card ?card1 ?waste)
    (forall ?card2
     (=>
      (and
       (deck_of_card ?card2 ?waste)
       (/= ?card1 ?card2))
      (> (coerce-to-integer (card_order_in_deck ?card1))
         (coerce-to-integer (card_order_in_deck ?card2))
      )
     )
    )
   )
   (Predicate)
   )
```

```
   )
  )
)
```

Si quisiéramos buscar la carta de un suit_stack (montón de cartas de un mismo palo en orden de valor comenzando por el as) pondríamos:

```
(exists ?card3
 (exists ?altc_stack
  (and
   (and
    (deck_of_card ?card3 ?altc_stack)
    (forall ?card4
     (=>
      (and
       (deck_of_card ?card2 ?altc_stack)
       (/= ?card3 ?card4))
      (>
       (coerce-to-integer( card_order_in_deck ?card3))
       (coerce-to-integer (card_order_in_deck ?card4))
      )
     )
    )
   )
   (Predicate)
   )
  )
 )
)
```

Ahora si lo que se desea es para una carta del waste llevarla a un suite stack, eso depende de las cartas disponibles del waste y del suit stack, por lo que pondríamos las restricciones de uno en el predicado del otro

```
(exists ?card1
 (exists ?waste
  (and
   (and
    (deck_of_card ?card1 ?waste)
    (forall ?card2
     (=>
      (and
       (deck_of_card ?card2 ?waste)
       (/= ?card1 ?card2))
      (> (coerce-to-integer (card_order_in_deck ?card1))
         (coerce-to-integer (card_order_in_deck ?card2))
      )
     )
```

```
    )
   )
   (exists ?card3
    (exists ?altc_stack
     (and
      (and
       (deck_of_card ?card3 ?altc_stack)
       (forall ?card4
        (=>
         (and
          (deck_of_card ?card2 ?altc_stack)
          (/= ?card3 ?card4))
         (>
          (coerce-to-integer( card_order_in_deck ?card3))
          (coerce-to-integer (card_order_in_deck ?card4))
         )
        )
       )
      )
      (Predicate)
     )
    )
   )
  )
 )
)
```

Si queremos que esto solo sea posible cuando la carta del waste sea de mayor valor que la del suit stack y del mismo palo, ponemos en el lugar del `(Predicate)` lo siguiente:

```
(and
 (= (value_number (card_value ?card3))
    (+ (value_number (card_value ?card1)) 1) )
 (/= (suit_color (card_suit ?card1))
     (suit_color (card_suit ?card3) )
)
```

Resultando en la siguiente restricción:

```
M_waste_to_column_stack
Indica si es posible mover una carta desde el Waste a un
Column Stack
(defrange ?card1 :FRAME Card cards_of_deck)
(defrange ?card2 :FRAME Card cards_of_deck)
(defrange ?card3 :FRAME Card cards_of_deck)
(defrange ?card4 :FRAME Card cards_of_deck)
(defrange ?waste :FRAME Waste)
(defrange ?altc_stack :FRAME Alternating_color_stack)
```

```
(exists ?card1
 (exists ?waste
  (and
   (and
    (deck_of_card ?card1 ?waste)
    (forall ?card2
     (=>
      (and
       (deck_of_card ?card2 ?waste)
       (/= ?card1 ?card2))
      (> (coerce-to-integer (card_order_in_deck ?card1))
         (coerce-to-integer (card_order_in_deck ?card2))
      )
     )
    )
   )
   (exists ?card3
    (exists ?altc_stack
     (and
      (and
       (deck_of_card ?card3 ?altc_stack)
       (forall ?card4
        (=>
         (and
          (deck_of_card ?card2 ?altc_stack)
          (/= ?card3 ?card4))
         (>
          (coerce-to-integer( card_order_in_deck ?card3))
          (coerce-to-integer (card_order_in_deck ?card4))
         )
        )
       )
      )
      (and
       (=  (value_number (card_value ?card3))
           (+ (value_number (card_value ?card1)) 1) )
       (/= (suit_color (card_suit ?card1))
           (suit_color (card_suit ?card3)) )
      )
     )
    )
   )
  )
 )
)
```

Esto sería mucho más fácil si el sistema soportara “patrones” o “templates” o “macros” (quizás “patrón” es más adecuado y “fashion”) de forma que se pudiera definir algo como:

```
(def-pattern
 (last-card-of-deck-satisfies
        ?card ?deck ?class ?predicate)
 (defrange ??cardX :FRAME ?class)
 (exists ?card
  (exists ?deck
   (and
    (and
     (deck_of_card ?card ?deck)
     (forall ??cardX
      (=>
       (and
        (deck_of_card ??cardX? deck)
        (/= ?card ??cardX))
       (>(coerce-to-integer (card_order_in_deck ?card))
         (coerce-to-integer (card_order_in_deck ??cardX))
       )
      )
     )
    )
    (Predicate)
    )
   )
  )
 )
)
```

donde ??cardX es una instancia de evaluación local al patrón que no es necesaria fuera del alcance de este. Y en este caso incluso se sugiere que la definición del rango de la variable pueda ser pasada como parámetro al patrón.

Entonces la definición de una carta que se puede llevar desde el waste hasta un suite stack sería de la siguiente forma:

```
(last-card-of-deck-satisfies ?card1 ?waste Card
  (last-card-of-deck-satisfies ?card2 ?suit_stack Card
    (and
      (= (value_number (card_value ?card2))
         (+ (value_number (card_value ?card1)) 1) )
      (/= (suit_color (card_suit ?card1))
          (suit_color (card_suit ?card2) )
    )
  )
)
```

Incluso se podría mejorar ya que la regla para poner en el suit stack es independiente del origen de la otra carta y se podría definir el patrón siguiente, nótese que sugerimos definir rangos dentro del patrón para los parámetros a efectos que se verifique qué tipo de elementos recibe el patrón

para evaluar, y por lo tanto cualquier subsunción de esa clase también cumple:

```
(def-pattern
 (satisfies-on-suit-stack ?card1 ?card2)
 (defrange ?card1 :FRAME Card cards_of_deck)
 (defrange ?card2 :FRAME Card cards_of_deck)
  (and
    (= (value_number (card_value ?card2))
       (+ (value_number (card_value ?card1)) 1) )
    (/= (suit_color (card_suit ?card1))
        (suit_color (card_suit ?card2) )
  )
)
```

Y la restricción anterior quedaría, con sus definiciones de rango, de la siguiente manera:

```
(defrange ?card1 :FRAME Card cards_of_deck)
(defrange ?card2 :FRAME Card cards_of_deck)
(defrange ?waste :FRAME Waste)
(defrange ?suit_stack :FRAME Suit_stack)

(last-card-of-deck-satisfies ?card1 ?waste Card
  (last-card-of-deck-satisfies ?card2 ?suit_stack Card
    (satisfies-on-suit-stack ?card1 ?card2)
  )
)
```

Si ahora quisieramos definir la restricción para un Alternating color stack simplemente la cambiaríamos para la clase que corresponde, de la manera siguiente:

```
(defrange ?card1 :FRAME Card cards_of_deck)
(defrange ?card2 :FRAME Card cards_of_deck)
(defrange ?waste :FRAME Waste)
(defrange ?altc_stack :FRAME Alternating_color_stack)

(last-card-of-deck-satisfies ?card1 ?waste Card
  (last-card-of-deck-satisfies ?card2 ?altc_stack Card
    (satisfies-on-suit-stack ?card1 ?card2)
  )
)
```

Esto permitiría facilitar el reuso de las reglas aprovechando las facilidades de herencia y uso de generalizaciones cuando sea posible. Incluso las reglas (restricciones) que definen en qué situación se generaliza (se aplica el patrón), podrían definirse dentro de los propios patrones.

Incluso, homogeneizando los términos y las reglas, este conjunto de patrones se podría utilizar para cualquier conjunto de objetos ordenados dentro de un contenedor, independientemente de la clase que representan.

Por otra parte reglas complejas son difíciles de leer y seguir. Esta fue otra de las razones para desarrollar IndentEd, para manejar texto anidado y presentarlo en múltiples formas y reglas de anidamiento. Esto facilitó mucho la verificación de formulas complejas ya que la herramienta Protégé dispone de muy pocas ayudas para el análisis de las fórmulas.

Además teniendo en cuenta las particularidades del ambiente en cuanto a la evaluación del las cláusulas, estos patrones permitirían el reuso de predicados, para las cuales ya está probada su evaluación en el sistema.

Ante las limitaciones del ambiente nos restringimos a hacer este proceso en forma manual para la definición de las restricciones del dominio de estudio.

Un trabajo que discute el tema de aplicación de patrones para el manejo de conocimiento en ontologías es **[SSTAAB01]** aunque su enfoque es más amplio, tratando de cubrir más aspectos de la ingeniería de ontologías y algo más complejo de aplicar en forma automática. Agregan un fuerte conjunto de restricciones de aplicabilidad (precondiciones y poscondiciones de las derivaciones que puede obtener la aplicación del patrón) para poder controlar la decidibilidad de sus ontologías. Se basan principalmente en definiciones en RDF-Schema aunque consideran variantes que permiten múltiples lenguajes de representación y de traducción. Al parecer versiones recientes de Ontoedit soportan ya este tipo de funcionalidades.

La propuesta presentada en este trabajo es mucho más humilde, pero al mismo tiempo más práctica y fácil de implementar y aplicar.

Multiplicidad de Reglas que controlan la representación.

A como está planteado el problema debemos hacer dos conjuntos de reglas.
El patrón `last-card-of-deck-satisfies` es una regla constructiva del Alternating color stack. Indica como se le agrega elementos y mantiene la propiedad de seguir siendo Alternating color stack. Mientras que la regla `alternating_color_stack_card` define que tipo de deck es aquel que puede ser considerado un Alternating color stack.

Sería deseable desde el punto de vista de las definiciones, que un solo conjunto de reglas definiera como se evalúan los objetos. Si se hace en forma constructiva tiene el problema que para evaluar si un deck puede ser un Alternating color stack, debo encontrar el conjunto de reglas de movimientos que aplicadas me determinen si el estado dado puede ser generado por esas reglas. Para estos conjuntos de cartas no es muy complejo, pero si pensamos en otros ejemplos como ser determinar si un cierto estado de un tablero de ajedrez es el resultado de un conjunto razonable de jugadas, quizás sea demasiado pedir a un sistema de razonamiento. Ahora si los objetos tienen definiciones que se pueden determinar sin información de la historia sobre como fueron construidos, simplemente evaluando su estado actual, un cambio en el conocimiento se reduce a preguntar si el objeto sigue manteniendo las propiedades de ser lo que era luego del cambio.

La otra opción es asumir que se debe poner varios tipos de reglas por eficiencia, que representen los cambios en el ambiente, la identidad y la clasificación, y asociarlas de alguna manera para que se mantengan vinculadas para informar al usuario del sistema.

Consultas Relacionadas a las Restricciones

Las consultas elaboradas a partir de restricciones que apuntan a la existencia de elementos que cumplan las propiedades necesarias para los movimientos permiten a su vez hacer consultas que indiquen las cartas que pueden participar del movimiento. Así si la posibilidad de realizar el movimiento M_turn_up_move_possible solo depende de si el stock está vacío no se puede modificar fácilmente para que indique la carta que está involucrada en este movimiento.

```
(exists ?stock
        (>    (coerce-to-integer    (number-of-slot-values
cards_of_deck ?stock)) 0))
```

Pero si en la restricción se verifica que exista una carta y no una propiedad genérica de los decks involucrados, entonces se puede modificar para convertirla en una consulta. Así la restricción M_waste_to_column_stack definida como sigue:

```
(exists ?card1
  (exists ?waste
      (and
        (deck_of_card ?card1 ?waste)…
```

Se puede transformar en la consulta de las cartas que cumplen para el movimiento M_waste_to_column_stack definida como sigue:

```
(findall ?card1
  (exists ?waste
      (and
        (deck_of_card ?card1 ?waste)…
```

Este simple cambio comanda encontrar todas las cartas que cumplen con la propiedad de poder ser movidas desde el waste al stock.

Otras sugerencias para mejora de la herramienta

Sería interesante que la herrmienta permitiera funciones de restricciones o consultas, por ejemplo, que permitiera “forall ?card that-satisfies-restriction_Z …” o “exists ?card in query_Y...”de forma de anidar restricciones y trabajar con subconjuntos que sabemos que cumplen ciertas propiedades. En parte se puede solucionar con el esquema de patrones propuesto, pero permitir esta funcionalidad ayuda al reuso de conceptos.

El sistema debería evaluar las restricciones que se cumplen para las clases con las que se está trabajando para asegurarse que no se ingresan restricciones que son redundantes entre sí. Incluso que el ambiente pueda identificar restricciones redundantes o equivalentes y que las optimice.

Las declaraciones del lenguaje se debieran tokenizar. Dentro de la base están explícitas en texto. Esto causa que si por alguna razón se debe cambiar el nombre de un slot o clase, eso no se ve reflejado en el texto de las restricciones, por lo tanto deben ser corregidas en forma manual o con un editor de texto dentro de los archivos del proyecto reemplazando cada aparición por el nuevo nombre.

Ayudaría el disponer de estructura de datos primitivas (TAD's) que manejen los contenedores clásicos como extensión a los slots que asocian a otras instancias. Esto resolvería situaciones como la de las *Card* que deben tener un *slot card_order_in_deck* para indicar el orden que tienen en un conjunto de cartas. El número de orden de una carta es una propiedad extrínseca a la carta, se da por una situación particular de la carta de pertenecer a un conjunto de cartas y estar en el orden dado. Pero no es atributo interno de la carta. Esto nos obliga a mantener ese *slot* y poner un conjunto de reglas que verifican la coherencia del orden de todas las cartas que pertenecen a un conjunto.

Habría que extender el lenguaje para que pueda actuar sobre la representación de la realidad, de forma que cambie *slots*, y clases entre eventos que se activen en la base. En los hechos una cierta funcionalidad de ejecución de código.

Referencias

[Stor1997] ACM Transactions on Database. Database Design with common sense business reasoning and learning. Veda C.Storey, Roger H.L.Chiang, Debabrata Dey, Robert C.Goldstein, Shankar Sudaresan. Volume 22, Issue 4 (December 1997).

[Broa2000] Andrew Broad. Case Based Reasoning for Code Understanding and Generation. Phd Thesis.

[Woo2000] Agent-Oriented Software Engineering: The State of the Art. Michael Wooldridge, Paolo Ciancarini. Eds. Lecture Notes in Computer Science 1957, First International Workshop, AOSE 2000, Ireland, June 2000, Springer-Verlag (Berlin).

[Con2000] Comparing Ontologies – Similarity measures and a Comparison study.

[Usc1998] An Experiment in Ontology Reuse, Proceedings of KAW'98, Eleventh Workshop on Knowledge Acquisition, Modeling and Management.Banff, Alberta, Canada, 1998. Mike Uschold, Peter Clark, Mike Healy, Keith Williamson, Steven Woods. Boeing Applied Research and Technology, Seattle, USA.

[Coh1999] Does Prior Knowledge Facilitate the Development of Knowledge-based Systems?. Paul Cohen, Vinay Chaudri, Adam Pease, Robert Schrag. Department of Computer Sciance, University of Massachusetts, 1999.

[Lev1966] Binary Code capable of correcting deletions, insertions and reversals. I.V.Levenshtein. Cybernetics and Control Theory, 10(8):707-710,1966.

[Broo1987] No Silver Bullet: Essence and Accidents of Software Engineering. Frederick P. Brooks. IEEE Computer, vol.20, no.4 (April 1987), pp10-19.

[Clan1993] Notes on "Heuristic Classification". William C. Clancey. Artificial Intelligence 59(1-2 February):191-196, 1993. Special Issue "Artificial Intelligence in Perspective".

[Cox1995] No Silver Bullet Revisited. Brad J. Cox. American Programmer Journal. November 1995.

[Cox1986] Object Oriented Programming: An evolutionary Approach. Brad J. Cox. Productivity Products Inc. Adison Wesley Publishing Company. Printed with Corrections April 1987.

[Ome22001] Syntactic-Level Ontology Integration Rules for E-commerce. Borys Omelayenko. 2001.

[Fell1998] Fellbaum C. 1998. WordNet: An electronic Lexical Database. The MIT Press.

[Pint2001] Helena Sofía Pinto & João P. Martins. 2001. A Methodology for Ontology Integration. K-CAP'01 ACM.

[Pin1999] Some Issues on Ontology Integration. H. Sofia Pinto, A Gómez-Pérez, J.P.martins. Proceedings of the IJCAII99's Workshop on Ontologies and problem solving methods: Lessons Learned and Future Trends.

[Boi2001] Ontologies and the Knowledge Acquisition Bottleneck. Mihai Boicu, Gheorghe Tecuci, Bodgan Stanescu, Gabriel C. Balan, Elena Popovici. Learning Agents Laboratory. Department of Computer Sciences. George Mason University. VA, USA.

[Ful1994] Why Post Industrial Society Never Came. Steve Fuller. ACADEME November-December 1994. pp 22-29.

[Mäd2000] Representation Language-Neutral Modeling of Ontologies. A. Mädche, H. P. Schnurr, S. Staab & R. Studer, Modellierung 2000.

[Gru1993] Tom R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. Padua Workshop on Formal Ontology, March 1993.

[Gua1995] Ontologies and Knowledge Bases: Towards a Terminological Clarification. Nicola Guarino, National Research Council, LADSEB-CNR, Pavoda Italy. Peirdaniele Giaretta. Institute of History of Philosophy, University of Padova, Italy. 1995.

[Fens2000] OIL in a Nutshell; D. Fensel; I Horrocks; F. Van Harmelen; S. Decker; M. Erdmann; and M. Klein.

[Omel2001] A Two-Layered Integration Approach for Product Information in B2B E-Commerce. Borys Omelayenko and Dieter Fensel.

[Izum2001] Building Business Applications By Integrating Heterogeneous Repositories Based on Ontologies, Noriaki Izumi and Takashira Yamaguchi.

[Nat2001] Ontology Development 101: A Guide to Creating Your First Ontology, Natalya F. Noy, Deborah L. McGuinness. Stanford University, CA, USA.

http://protégé.stanford.edu/publications/ontology_development/ontology101.html
[Uschol96] Building Ontologies: Towards a Unified Methodology. Mike Uschold. AIAI-TR-197. September 1996. University of Edimburg.
[KIF1998] M.R.Genesereth. Knowledge Interchange Format. Draft Proposed American National Standard (dpans). ncits.t2/98-004. http://logic.stanford.edu/kif/dpans.html. http://www-ksl.stanford.edu/knowledge-sharing/kif/
[Gin1991] Knowledge Interchange Format. The KIF of Death. M.Ginsberg. AI Magazine, 5(63), 1991.
[Moy1991] John A. Moyne. LISP: A first language for computing. Van Nostrand Reinhold. NY. USA, 1991.
[Aho1986] Compilers: Principles, Techniques and Tools. Chapter 5. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Bell Telephone Laboratories. Addison Wesley Publishing Company, MA, USA, 1986.
[Protégé] Protégé web site: http://protégé.stanford.edu.
[Noy2000] N. F. Noy, R. W. Fergerson, & M. A. Musen. The knowledge model of Protege-2000: Combining interoperability and flexibility. 2th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000), Juan-les-Pins, France, . 2000.
[PALplug] PAL plugin. PAL Constraints and Queries Tabs. http://protege.stanford.edu/plugins/paltabs/PAL_tabs.html. The Protégé Axiom Language and Toolset ("PAL") http://protege.stanford.edu/plugins/paltabs/pal-documentation/index.html.
[SSTAAB01] Engineering Ontologies using Semantic Patterns. Steffen Staab, Michael Erdmann, University of Karlsruhe, Ontoprise. Alexander Maedche, FZI Research Center for Information Technologies. Karlsruhe, Germany. IJCAI-01 Workshop on Ontologies and Information Sharing, August 2001, Seattle USA.