Capítulo 2

Métricas y estándares de comparación

En este capítulo se proporcionará una descripción del estado del arte en materia de métricas y evaluaciones aplicadas a sistemas de programación lógica incluyendo aquellos que soportan la semántica de *answer set*.

Una métrica de software es la proyección de un conjunto de objetos en el mundo de la ingeniería de software a un conjunto de objetos en el mundo matemático [R2.1]. Los objetos en el mundo del software pueden ser propuestas, productos y procesos. Los objetos en el mundo matemático pueden ser números o vectores de números. Este mapeo puede ser definido en distintas escalas tales como nominal, ordinal, intervalo, etc.
Una métrica es usada para caracterizar cierta propiedad de un objeto o una clase de objeto; en la ingeniería de software, esta caracterización es cuantitativa, por ejemplo la métrica "líneas de código" caracteriza la propiedad de tamaño de un código fuente asociando un número con él.
[R2.1] El aplicar una métrica a un objeto de software debe alcanzar ciertas metas predefinidas, las cuales cambian de acuerdo a 5 características importantes:
1. Que objetos en la ingeniería del software están siendo medidos.
2. Porque están siendo medidos o cual es el propósito.
3. Quién esta interesado en estas métricas.
4. Cuales de sus propiedades están siendo medidas.
5. En que ambiente los objetos son medidos.

2.1 Métricas

Las métricas de software nos permiten estimar costo y esfuerzo en realizar un proyecto, evaluar la calidad del software realizado, predecir el tiempo invertido en mantenimiento de un programa o validar las mejores prácticas para el desarrollo del mismo.
A diferentes características que nos interesen es la métrica empleada para obtenerla, por ejemplo si deseamos estimar el costo y el esfuerzo entonces se necesita conocer el tamaño de un programa, si lo que se requiere es medir la complejidad, existen diversidad de métricas como McCabe y Halstead [R2.16], por lo que no siempre una sola métrica nos da toda la información que necesitamos.
Para los sistemas propuestos, en el presente trabajo nos interesa medir:

- Su poder de representación, expresividad y sencillez para la enunciación de problemas de optimización.
- La complejidad de un programa.
- La eficiencia en el cálculo de soluciones.
- La confiabilidad en resultados.

Moores en 1998 [R2.2] propone 4 tipos de métricas aplicadas a códigos en lenguaje Prolog, que intentan cuantificar la complejidad de un programa en términos de medir la dependencia entre las reglas; éstas son: número de líneas de código (LOC), número de

reglas (NR), número de nodos que representen predicados únicos (UPN) y modelo de complejidad ciclomática de McCabe v(G). LOC y NR son métricas que intentan relacionar el tamaño de un programa con la complejidad del mismo. UPN y v(G) requieren que el programa sea reducido a alguna forma de representación desde la cual, el comportamiento del programa pueda ser fácilmente visto. De éste concepto, surge la idea de usar una red gráfica de nodos y arcos. El argumento detrás de esta aproximación es que, la complejidad del grafo resultante sea ligada con algún indicador de la tendencia de error en un programa, para con ello calcular cual es la probabilidad de que una pieza de software falle.
Otra razón, es que la tendencia del programador a cometer errores crece a medida que un código es modificado, la complejidad del programa se eleva y el usuario pierde la perspectiva de cual pieza de código afecta a otra, dando como resultado errores inesperados.
No es de extrañarse que gran parte de la investigación reciente, se haya enfocado en usar la representación por medio de grafos para caracterizar alguna propiedad de los sistemas basados en reglas. Por ejemplo definir un grafo para representar las llamadas de una regla a otra o ayudándose de él para describir su diagrama de flujo, capturando ciertos atributos del grafo, como número de nodos, profundidad, anidamiento, longitud de los caminos y otros.
Los autores realizaron un ejercicio aplicando estas métricas a 80 programas en Prolog, divididos en aquellos que contienen errores y sin ellos; tomando en cuenta diversas condiciones externas, concluyen que un programa en Prolog es calificado como más complejo y la tendencia de error aumenta cuando el número UPN excede una cota de 35+-5 predicados por programa. Dado que cada predicado único representa al menos un camino lógico trazado por el programador, a mayor número de caminos, más complejo es el programa.

En la misma línea pero para programas lógicos concurrentes Zhao et al. [R2.3] proponen una métrica que mida complejidad, llamada ADN (*argument dependence net*). Esta métrica calcula la dependencia entre los argumentos en un programa; pero aplicarla significaría tener un conocimiento profundo del lenguaje usado, cómo se realizan las llamadas a las funciones o predicados y cómo se intercambian los datos entre ellos.

Por otra parte, a pesar de que la importancia de los lenguajes basados en reglas es ampliamente reconocida, ya que ofrecen muchas ventajas para representar el conocimiento humano, los estudiosos del tema reconocen que la programación en este tipo de lenguajes, algunas veces es dificil de hacer e incluso de modificar y mantener por la falta de constructores de control y la casi total dependencia de los datos, que en tiempo de ejecución determina la secuencia a seguir [R2.4].

De ahí el interés creciente en medir la complejidad conceptual, entendiéndose ésta, como el predecir el esfuerzo humano tanto de tiempo como de costo para entender y elaborar un programa. Es un hecho que es dificil tratar de medir o cuantificar conceptos tales como esfuerzo, por lo que este campo es aún controversial, incluso en la programación procedural tradicional.
En ésta última área, las métricas para evaluar la complejidad conceptual se agrupan en métricas de volumen, que están interesadas tanto en medir el tamaño físico de un programa como en cuantificar la forma en que el control del mismo pasa de principio a fin, midiendo los tipos y cantidad de ciclos, ramificaciones y condiciones usadas; otro

grupo es el de las métricas interesadas en el flujo de datos, esto es, medir el tipo y cantidad de información que es transferida entre los componentes de un programa.
O'Neal y Edwards en 1994 [R2.4] sugieren aplicar un conjunto de métricas divididas en dos grupos: métricas de volumen, iguales a las aplicadas para programas procedurales y métricas de reglas, que cuantifican la complejidad de las reglas individuales, midiendo el tamaño de la regla como el número de objetos conceptuales únicos usados en ella, y midiendo el número de interacciones entre las mismas reglas y los datos. Sus resultados sugieren que la dependencia entre las reglas es un elemento significativo de la complejidad de un programa.
Reconocen que el analizar un programa en una implementación particular de este tipo de lenguajes no es recomendable por la variación en semántica y sintaxis que obscurece los factores que contribuyen a la medición de la complejidad, por lo que proponen diseñar un lenguaje independiente, al cual se convertirán los programas de los lenguajes a evaluar y las mediciones se aplicarán al programa representado en este lenguaje intermedio.

Otro punto de vista define a la complejidad como el esfuerzo requerido en realizar pruebas de un programa, *p-complexity* [R2.5] cuya métrica es el número total de casos de prueba requeridos para probar la exactitud de un camino continuo en un programa. Ellos afirman que la efectividad de esta métrica es empírica, ya que varía de un experimento a otro.

2.2 Estudios comparativos

Otra forma que la comunidad de programación lógica ha encontrado para establecer lineamientos en el área de métricas, es la realización de pruebas comparativas (*benchmarks*) entre diferentes sistemas, pero se ha reducido a solamente preocuparse por el rendimiento, dejando de lado cuestiones como la eficiencia, facilidad y expresividad que ofrece un sistema para modelar un problema dado, o la dificultad en el mantenimiento de un programa.

La importancia al realizar estudios comparativos entre diferentes sistemas o lenguajes, radica en que estos, proveen un gran cúmulo de información y dan lineamientos para futuros desarrollos, puede mostrar cual lenguaje es el mejor para una tarea en particular, por ejemplo, aquel más adecuado para introducir a un programador novato en el área. Dado el gran avance en el área de ASP y el creciente interés por desarrollar nuevos sistemas que soporten esta semántica, futuros desarrollos deben darse bajo ciertas directrices que faciliten este trabajo y permita evaluarlos de forma objetiva y eficiente.

Un ejemplo de lo anterior, es el ejercicio comparativo que se realizó en el taller de razonamiento no monotónico [R2.6] entre diferentes sistemas que implementan la semántica de *answer set*; algunos de los más destacados fueron DLV, Smodels, ASSat, cmodels, aspps. Ahí se propusieron lineamientos [R2.7] para realizar este tipo de pruebas, entre las que destacan, medir correctamente el tiempo de ejecución, asegurarse que los resultados obtenidos puedan ser reproducibles por terceros, facilitando la mayor cantidad de información posible acerca del tipo de hardware y software utilizado, las instancias que se usaron, como generarlas y porque fueron escogidas, para que finalmente los resultados mostrados sean confiables.
Este tipo de comparaciones no es trivial, debe tomarse en cuenta que el tiempo invertido es significativo, principalmente por la generación de los casos de prueba, por lo que

usualmente se automatiza utilizando otras herramientas, entre las que destacan GraphBase y TheoryBase.
Es perceptible que el interés de este tipo de ejercicios, se centra en el rendimiento de los sistemas, por lo que sea cual sea el problema propuesto, las instancias deben darse en gran cantidad para que la medición del tiempo de ejecución sea calculable.

Una iniciativa interesante es la que propone Schaub [R2.8] la cual sugiere crear una infraestructura que permita establecer estándares de comparación, que controlaría y almacenaría diversos problemas, además de proporcionar el medio ambiente computacional para que los sistemas sean ejecutados en igualdad de condiciones. Reconoce que la dificultad del trabajo radica en que no existe una estandarización en los lenguajes, por lo que la funcionalidad difiere uno de otro, así que adicionalmente fue necesario categorizar los problemas para poder evaluar diferentes particularidades de los sistemas.

Adicionalmente cada grupo de investigadores, valora desde su perspectiva, las características de su sistema. Por ejemplo los creadores de DLV realizaron un estudio comparativo[R2.9] con Smodels y DeReS, desde el punto de vista cualitativo y cuantitativo. Cualitativamente comparan el poder de representación del sistema, esto es la habilidad para describir un problema en el lenguaje dado en una forma sencilla y natural; cuantitativamente se refieren al rendimiento. Sus resultados muestran que DLV es por mucho superior a sus competidores Smodels y DeReS. Usaron una colección diversa de problemas entre los que destacan *Ancestor, Graph 3-Colorability* y *Compañías estratégicas*.
Este último ejemplo es de especial interés, ya que aseguran [R2.9] que los sistemas como Smodels cuya programación es normal, no pueden representarlo de una forma general, solamente creando programas a la medida para instancias particulares, cuya implementación requiere gran inversión de tiempo. DLV fue capaz de hacerlo gracias a que soporta disyunción, aunque su rendimiento disminuyó con cierto tipo de instancias.
La descripción del problema se encuentra en [R2.10] y en el capítulo 3 de este documento. El código en DLV se ve así:

```
% As we want to produce X, Y or Z must be strategic.X is produced by Y or Z
strategic(Y) v strategic(Z) :- produced_by(X,Y,Z).

% W is strategic, if it is controlled by strategic companies X, Y, and Z
strategic(W) :- controlled_by(W,X,Y,Z),
        strategic(X), strategic(Y), strategic(Z).
```

En el manual de *lparse* [R1.10], se menciona que usando utilerías adicionales al sistema es posible calcular modelos estables de programas lógicos disyuntivos. Sin embargo ésta funcionalidad es aún primitiva y smodels no lo hace correctamente.
Parte del código en Smodels es así:

```
{strategic(Y), strategic(Z)} :- produced_by(X,Y,Z).
strategic(W) :- controlled_by(W,X,Y,Z), strategic(X), strategic(Y), strategic(Z).
```

Otro ejemplo clásico, frecuentemente utilizado es *Graph 3-Colorability* cuya implementación en DLV es la siguiente:

Col(X,r)v Col(X,g) vCol(X,b) :- node(X).
:- edge(X,Y), col(X,C), col(Y,C).

Para el caso de Smodels el código luce así, ya que no se puede usar disyunciones.
Col(X,r) :- node(X), not col(X,g), not col(X,b).
Col(X,g) :- node(X), not col(X,r), not col(X,b).
Col(X,b) :- node(X), not col(X,r), not col(X,g).
Bad :- edge(X,Y), col(X,C), col(Y,C), not bad.

En este caso, que estan interesados en medir el poder de representación del sistema, el equipo de DLV no hace uso de ninguna métrica cuantitativa para caracterizar esta propiedad, simplemente hacen la siguiente afirmación *"Evidently, DLV allows for a more elegant problem representation"* [R2.9] ; lo que nos muestra que aún se usa una forma intuitiva para medir tal característica.

También en el trabajo mencionado se realizaron pruebas para medir el rendimiento de un programa en el caso donde un problema es resuelto usando funciones de agregación y en otro caso no las usa. Sus resultados muestran mejoras considerables en sus tiempos de ejecución cuando las funciones de agregación son usadas. A su vez también compararon DLP^A con Smodels y observaron similitudes entre sus funciones de agregación, aunque algunas aún no existen en Smodels como *#min #max y #times* .

El equipo de DLV ha sido de los promotores para realizar estudios comparativos [R1.5]. Su problema ha sido encontrar sistemas similares que soporten programación lógica disyuntiva. Finalmente consideraron a Smodels, ASSAT y GnT y usaron una amplia gama de ejemplos de diversos dominios y complejidad, buscando que la programación de estos, sea lo más sencilla posible.

Smodels ha sido comparado [R1.9] con otros sistemas similares como SLG. En estas pruebas, fue usado TheoryBase como generador de grafos, los casos se generaron aleatoriamente, y los problemas fueron seleccionados principalmente del área de combinatoria, como por ejemplo *3-Coloring y Hamiltonian path*. Su interés se centró en determinar la escalabilidad y límites de la implementación, por lo que en su juicio los problemas de combinatoria son lo suficientemente complejos y de gran tamaño para medir tales características. En segundo evaluaron su estabilidad en cuanto al poder declarativo, apoyándose de la generación aleatoria de casos de prueba y cambiando el orden de las reglas en los programas, ya que algunas veces este orden puede mejorar o menguar el rendimiento de un sistema. Finalmente usando referencias que son independientes de la implementación, como las instancias más dificiles y la eficiencia del mejor algoritmo en este tipo de problemas, les permitió medir la eficiencia y el costo de implementación de su sistema.

2.3 Herramientas para automatización de pruebas

Una propuesta interesante es la que ofrecen los creadores de TheoryBase[R2.11], el cual surgió de la necesidad de crear un medio de experimentación para el sistema DeReS (*Default Reasoning System*).
DeReS[R2.12] es la primera implementación de la Lógica por Default de Reiter, como de teorías por *default* y modelos estables de programas lógicos.

Debido a la falta de ambientes y elementos para pruebas, se creó una herramienta que fuera capaz de generar instancias de forma fácil, realistas y con significado. Esta es TheoryBase, cuya salida es fácilmente reproducible y distribuida, de tal forma que también otros sistemas puedan usarla. El método de generación es controlado por parámetros, lo cual permite a los usuarios generar diferentes casos de prueba.
TheoryBase usa como entrada los grafos generados por la herramienta Stanford GraphBase creado por Donald E. Knuth y da como resultado la codificación de los grafos en términos de teorías por default y programas lógicos.
GraphBase es una colección de conjuntos de datos y procedimientos los cuales generan grafos y varios programas de demostración. Permite a los usuarios generar familias de grafos, directos, indirectos, con peso, regular, aleatorios, etc. Por ejemplo puede generar el grafo de las distancias entre 128 ciudades de Norteamérica.
GraphBase se encuentra públicamente disponible y es ampliamente usado para el trabajo de experimentación que requiere grafos. Una característica importante es que cada grafo generado tiene un identificador único, el cual permite guardar y reproducir fácilmente diferentes casos de prueba. Varios ejemplos pueden ser hallados en [R2.13].
De entre los principales problemas que TheoryBase codifica son:
Graph coloring, hamiltonian cycles, maximal independent sets, maximal matchings, existencia de kernels.
TheoryBase también asocia un identificador a cada teoría que genera, de tal forma que facilitan el intercambio de ejemplos. Esto es una extensión del mismo concepto usado en GraphBase.
Un identificador de TheoryBase es un par (α, β) donde α es el identificador de GraphBase y β se refiere a la traducción a TheoryBase, como pueden ser coloring, kernels, etc.
La semántica de un identificador (α, β) esta dado por un programa lógico o una teoría por default, obtenida por el siguiente proceso:

1. Generar un grafo G desde GraphBase identificado por α, el cual codifica el método usado para crear dicho grafo.
2. Producir un programa lógico o una teoría por default aplicando la traducción β al grafo G.

Algunos ejemplos pueden ser encontrados en [R2.11]

Existen otras aproximaciones a TheoryBase, como son las bases de datos de ejemplos, las cuales son comúnmente creadas para experimentación con algoritmos de programación lineal y en otras numerosas áreas. Otra opción frecuentemente usada es la generación aleatoria de datos, que aunque ofrece un número ilimitado de casos de prueba, está documentado, que genera propiedades que raramente ocurren en la vida real.
Ninguna de estas aproximaciones ha sido completamente desarrollada para experimentación con programación lógica o razonamiento no monotónico.

Un ejemplo de una implementación de base de datos de programas de prueba es la herramienta *"Benchmark generator"* [R2.14], la cual genera programas lógicos en diferentes implementaciones como son: noMoRe, Smodels, DLV y quip, abarcando problemas de planeación y los que pueden ser representados por grafos.
La herramienta es implementada en ECLiPSe[R2.15], un sistema de software para el desarrollo de aplicaciones programadas con restricciones, en el área de planeación, planificación de horarios, asignación de recursos, transporte, etc. Es también ideal para

la enseñanza de técnicas de solución de problemas de combinatoria, como pueden ser, modelado, programación con restricciones y programación matemática.
"Benchmark generator" usa opciones de menú con las cuales, el usuario elige que tipo de grafo desea componer: *complete, simplex,* etc. selecciona el problema: *graph coloring, hamiltonian cycles, independent sets*, y finalmente el lenguaje de codificación: DLV, Smodels o quip.
Esta herramienta es un intento por estandarizar la generación de casos de prueba, de varios problemas típicos, pero para nuestro caso de estudio no fue útil ya que no incluye ejemplos de optimización.

Resumen del capítulo

En este capítulo se presentaron diversidad de aproximaciones en el uso de métricas a programas lógicos, las cuales surgen a partir de lo ya experimentado en el ambiente de la programación procedural, pero aún no existe, hasta el momento, una propuesta para programas *answer sets*. De hecho, en ASP, se han enfocado en realizar comparaciones de rendimiento entre sistemas y no existe una estandarizacion ni en métodos, tipos de problemas o generación de instancias.
El siguiente capítulo muestra los ejemplos que se implementaron en los sistemas que abarcan nuestro estudio.