Capítulo 4

Resultados

Este trabajo inició identificando las características más significativas de algunos sistemas ASP, referentes a su capacidad en modelar problemas de optimización y en el uso de funciones de agregación; después se realizó una búsqueda en publicaciones para extraer información tanto de métodos de medición de software como de estudios comparativos aplicados a sistemas tradicionales y a sistemas basado en reglas, de donde obtuvimos que el trabajo realizado en el área de la programación declarativa no es suficiente y menos aún en el área de ASP, cuyo interés se centra en medir el rendimiento de los sistemas que soportan esta semántica.
Posteriormente se realizó una búsqueda exhaustiva de ejemplos, que mostrarán la robustez de las características estudiadas; tanto esta búsqueda como las pruebas realizadas requirió esfuerzo importante, debido, a que todavía es un área de investigación reciente y no hay suficientes registros de trabajos preliminares.
Finalmente se eligieron y aplicaron diversas métricas a estos ejemplos cuyos resultados proporcionan información para calificar la complejidad, confiabilidad, rendimiento y uso de cada sistema y demostrar que aunque cuantiosas métricas han sido ampliamente probadas en ámbitos distintos al de la programación lógica, en el área de ASP el trabajo está en su etapa inicial.

La siguiente tabla resume las características significativas encontradas en cada sistema:

Característica	**A-POL**	**DLV**	**Smodels**	**Aset-Solver**
Disyunción	soporta cláusulas disyuntivas	sentencias disyuntivas	no acepta sentencias disyuntivas	no acepta sentencias disyuntivas
Optimización	define funciones en términos de desigualdades	weak constraint	minimize, maximize	min, max
Agregación	add, sum, mul	min, max, count, sum, times	card, sum	card, sum
Extras	Recuperación de testigo, orden parcial, determina que átomos se desean ver en la salida: *focus*	En la línea de comando se elimina la salida de los hechos: *nofacts* . Uso de filtros en la salida.	Determinar el número de modelos estables a ser calculados: *compute*	Muestra en la salida solo los átomos de interés: *show*

Se observa que todos cuentan con funciones de agregación y diferentes formas de obtener modelos óptimos.

De las métricas aplicadas a los sistemas se esquematizan los siguientes resultados:

Característica	**A-POL**	**DLV**	**Smodels**	**Aset-Solver**
Eficiencia	SI	NO	SI, excelente	

Confiabilidad	No siempre	SI	Genera mucha basura	
Poder de representación	SI Facilidad para representar problemas propuestos de forma matemática.	SI Representación usando disyunción	SI Restringido a usar representación normal	NO Etapa muy primitiva , solo ejemplos pequeños
	Interés en el camino para obtener el valor óptimo	optimización de problemas	optimización de problemas	Poco o ningún soporte
	Uso de orden parcial	fácil uso de funciones de agregación	etapa primitiva sus funciones de agregación	
Complejidad	Alta	Baja	Media	
Nivel de configuración del sistema	Medio	Muy fácil	Dificil	Dificil

La investigación realizada en el campo de métricas y estudios comparativos se reúne en la siguiente tabla. Cabe hacer la observación que no fueron consideradas aquellas métricas que son aplicadas durante las diferentes fases de elaboración de un software, análisis, diseño, etc., porque nuestra investigación considera a los sistemas estudiados como cajas negras, sin involucrarse en el diseño y construcción de los mismos.

	Moores	**Moores**	**O´Neal**	**Tsai**	**Estudios comparativos**
Tipo de métrica	De complejidad textual, aquella que esta basada en el tamaño del programa.	De complejidad estructural, basada en las estructuras de control del programa.	De complejidad textual y estructural.	Dinámica, basada en el número de pruebas realizadas a un programa.	
Propiedad a medir	Complejidad y medición de las tendencias de error en un programa.		Complejidad conceptual o el esfuerzo humano en entender un programa.	La complejidad o el esfuerzo requerido en probar un programa.	Rendimiento dado con un número significativo de hechos.
Métrica	Número de líneas de código. Número de reglas.	Número de nodos que representen. Predicados únicos.	Mide la complejidad de la regla como el número de objetos usados en ella. Número de líneas de código. Número de interacciones entre reglas y datos.	Número de casos de prueba requeridos para probar un programa.	Obtener el tiempo de ejecución de un programa. Categorizar en número y tipo las instancias usadas. HW y SW utilizado.

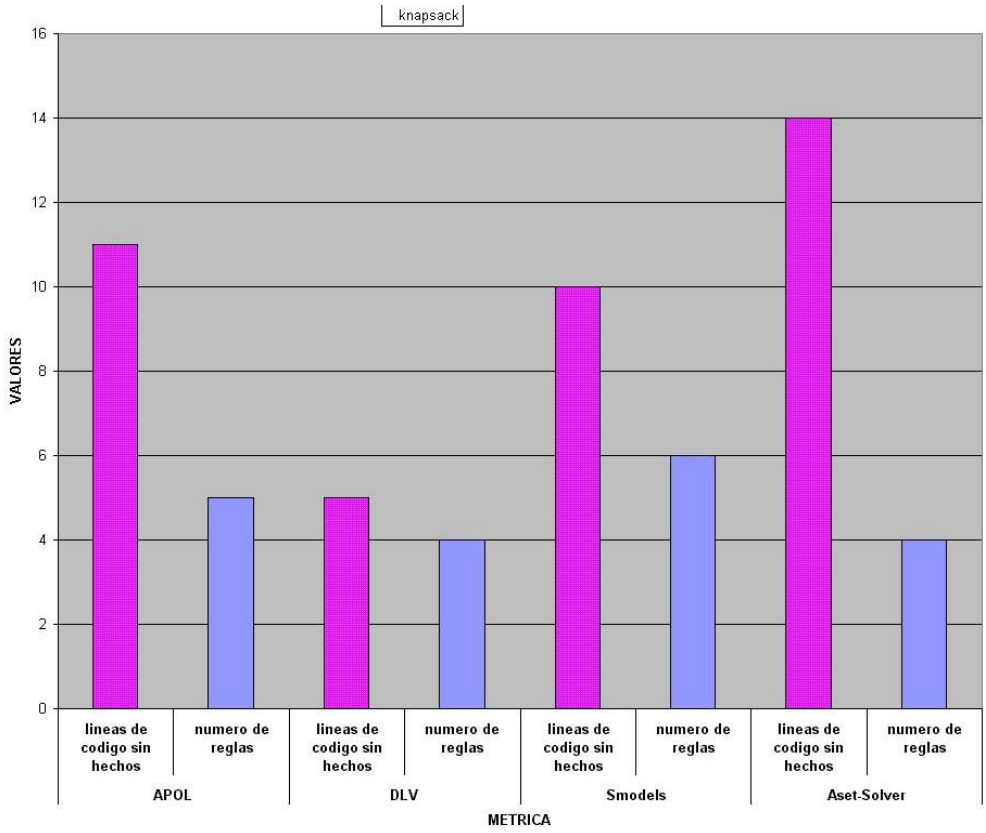
Propuesta y método	Mostrar diferentes códigos a diversos usuarios y medir el tiempo que tarda en comprenderlo.	Reducir el programa a su forma gráfica para contar el número de nodos y arcos.	Diseñar un lenguaje intermedio al cual se convertirán los lenguajes evaluados. Implementar las herramientas de conversión.	Herramientas automatizadas que ejecuten los diversos casos de prueba en un programa.	Uso de herramientas para la generación de instancias, como TheoryBase y GraphBase. Creación de una base de datos de problemas de diferentes tipos.

El tamaño de un programa parece ser una constante en estos estudios al relacionarlo directamente con la complejidad del mismo.
Aplicando algunas métricas a los ejemplos propuestos, se muestran los siguientes resultados:

SISTEMA	MÉTRICA	3.1 Mochila binaria	3.2 Fast Food	3.3 Camino mas corto	3.4 Cartoon	3.5 Subasta de objetos	3.6 Min. árbol ab.	Min. árbol ab. c/ agregación	3.7 Formación de equipos
A-POL	líneas de código	11	19	4	21				
	número de reglas	5	3	2	6				
DLV	líneas de código	5	17	4	7	7	7	4	6
	número de reglas	4	15	3	7	2	6	31	6
Smodels	líneas de código	10				8	16		
	número de reglas	6				3	9		
Aset-Solver	líneas de código	14							
	número de reglas	4							

Se calculó tanto el número de líneas de código como de reglas, sin contar los hechos, ya que un programa se entiende por su lógica no por los datos involucrados.
Se puede observar que para el ejemplo 3.4, el uso de funciones de agregación en DLV permite facilitar la programación con respecto a A-POL que no las soporta. También el problema del mínimo árbol abarcador, 3.6, fue probado sin y con funciones de agregación, siendo ésta última, la que mejora en mucho la lectura y redacción del mismo.

Para el ejemplo 3.1, de la mochila binaria, DLV mostró ser el más compacto, y podemos asegurar que no es el más complejo La desventaja en A-POL fue la dificil interpretación de las funciones de biblioteca que usa, por lo que tuvo que redefinirse una de ellas para evitar los errores que mostraba el compilador. Gráficamente se muestran las cantidades en líneas de código de este ejemplo:



Con respecto al rendimiento, se probó el código con diferente número de instancias:

Sistema	Métrica	Mochila	Binaria
		49 instancias	**70 instancias**
A-POL	tiempo	35.3 seg.	1:38.9 m.
	modelos resultantes	1 (el 5º. de dlv)	1
DLV	tiempo	22:26.3 m.	1:29:10.4 hr.
	modelos resultantes	6, todos óptimos	6
Smodels	tiempo	**0.3 seg.**	**0.7 seg.**
	modelos resultantes	37 (solo genera un óptimo, el 2º de dlv)	29

Los resultados fueron bastante interesantes, primero, como se mencionó anteriormente, la implementación en DLV es la más compacta, sencilla y entendible y con este ejercicio afirmamos, que es el único sistema que genera la salida correcta, con los 6 modelos resultantes óptimos. Sin embargo, su rendimiento es pobre comparado con el resto de los sistemas.
Por otra parte, A-POL solamente genera 1 modelo correcto, aunque demuestre ser más rápido, no es lo suficientemente confiable. Además pequeños cambios que se le hicieron al código provocó la generación de errores que no pudieron ser eliminados.

El comportamiento de Smodels es extraño, su rendimiento es excelente pero la generación de sus modelos es diferente a lo que se espera, Smodels obtiene un modelo que cumpla con las restricciones y partir de él, busca otro mejor y así sucesivamente, hasta llegar al último modelo el cual es el óptimo, por lo que si existen más –como en este caso- ya no los busca.
Se verificó este comportamiento con el experto del sistema Patrik Simons [R1.8], quien comentó que solo modificando el código fuente de Smodels, este generaría los 6 modelos óptimos. Se comprobó está información bajando, modificando y compilando *smodels* y probando nuevamente el ejemplo con los siguientes resultados:

Sistema	**Métrica**	**Mochila**	**Binaria**
		49 instancias	**70 instancias**
Smodels modificado	tiempo	0.6 seg.	0.6 seg.
	modelos resultantes	79 incluyendo 6 óptimos	23

En cuanto al problema 3.3 a continuación se muestran las pruebas realizadas, mostrando que A-POL supera en rendimiento a DLV.

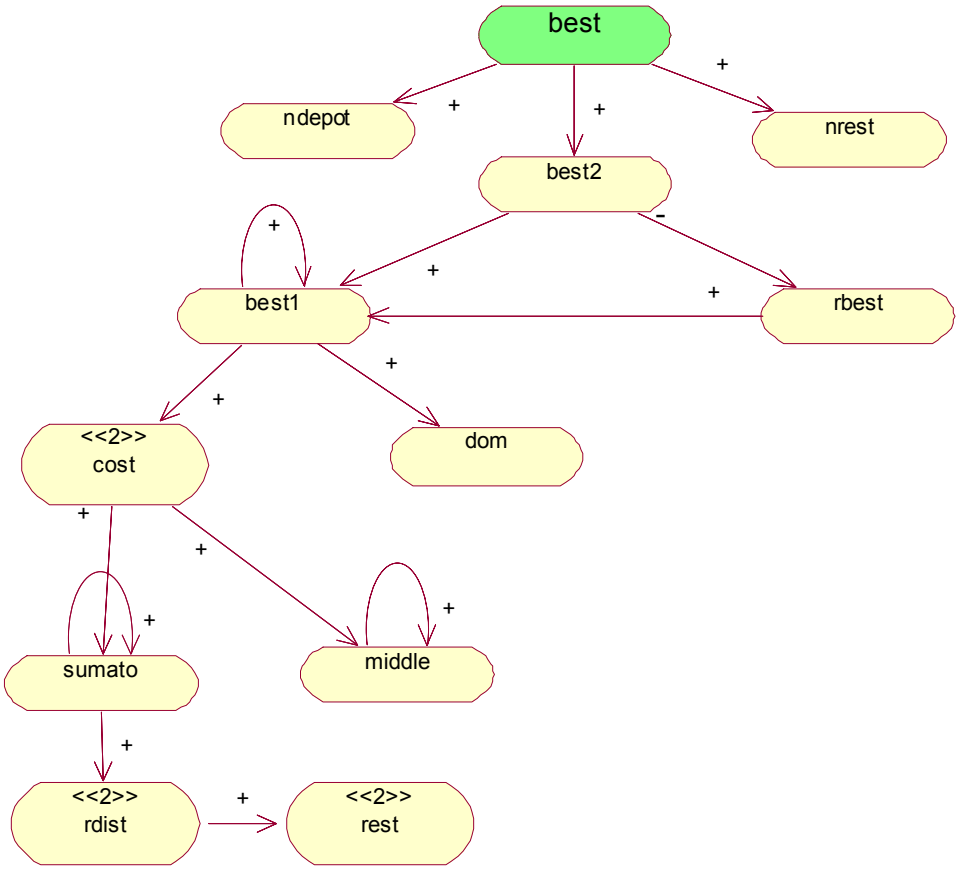
Sistema	**Métrica**	**Shortest path 20 v y 400 arcos**	**Shortest path 60v y 3600 arcos**
A-POL	tiempo	**19.8 seg.**	**12:38.4 m.**
DLV	tiempo	24:52.8 m.	1:44:13.0 ++

En [R1.1] también se reportan resultados similares a los aquí obtenidos.

Con respecto al problema 3.2, fastfood, los resultados fueron los siguientes:

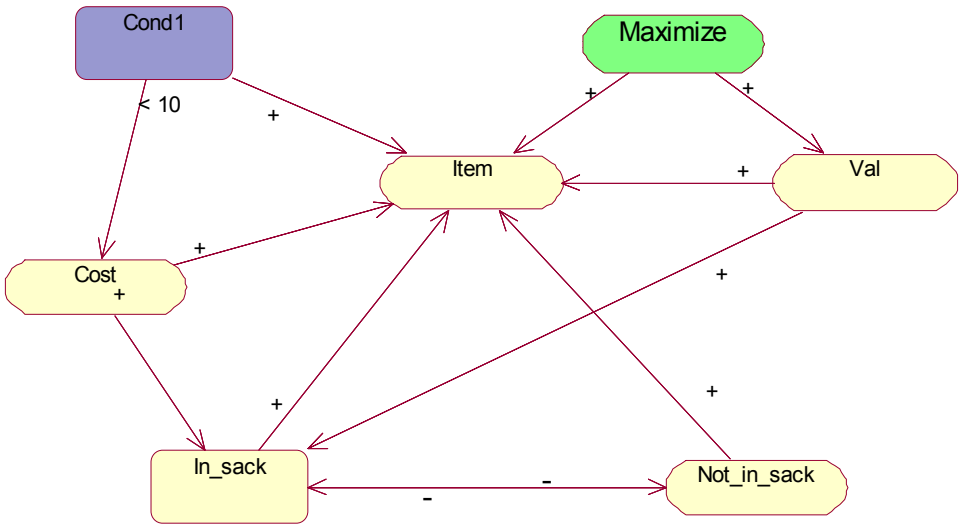
Sistema	**Métrica**	**Fastfood usando 50 instancias**
A-POL	tiempo	**9:24.4 min**.
DLV	tiempo	58:47.2 min.

Podemos observar que el rendimiento de DLV es pobre con respecto a A-POL. Probablemente porque el código en DLV no es el más óptimo, además que fue dificil de entender, por lo que se uso la representación gráfica para distinguir las llamadas entre predicados y visualizar su nivel de complejidad.



Podemos observar que la complejidad del código es directamente representada en el grafo, el cual muestra 5 niveles y las llamadas recursivas de algunos predicados. Sin la ayuda del grafo, a simple vista, es dificil definir en el código cual predicado genera la salida esperada.

Sin embargo esta forma de representación no siempre es aplicable en todos los casos, por ejemplo, el siguiente grafo de dependencia del problema de “la mochila binaria” en Smodels, nos haría pensar que es complejo o dificil de leer, pero revisando el código, nos damos cuenta que no es así.



Para el resto de los ejemplos se sintetizan los resultados en la siguiente tabla:

	A-POL	DLV	Smodels
Subasta de objetos	NO SE IMPLEMENTÓ	1 modelo óptimo sel{a}	Muestra **más** de un modelo.
Compañía Cartoon Co.	**Código incorrecto**, no muestra el modelo completo. La versión anterior 1.0b sin problemas	Fué fácil de entender el código que usa funciones de agregación	NO SE IMPLEMENTÓ
Mínimo árbol abarcador	NO SE IMPLEMENTÓ	Implementado con y sin funciones de agregación. **Errores** en el código tomado del tutorial	Se programó usando el código en DLV
Formación de equipos		Muestra la facilidad de uso de las funciones de agregación	NO SE IMPLEMENTÓ
Compañías estratégicas		La representación es sencilla usando disyunción	**No es posible representarlo**, incluso usando las sugerencias de los diseñadores

Otras observaciones importantes son:

Smodels tiene una diversidad de opciones para introducir en la línea de comandos, las cuales al ser probadas provocaron que la salida de los modelos no siempre fuera la misma, esto inhibió nuestra confiabilidad en los resultados obtenidos. Además su comportamiento cambia al usar sus funciones de optimización (*maximize, minimize*) ya que al encontrar un modelo lo imprime, no importando si es el óptimo y a partir de él busca "los mejores". También comprobamos que se debe tener mucho cuidado al usar estas instrucciones debido a que su sintaxis nos confundió con respecto al uso de llaves ´{´ o corchetes ´[´, provocando la salida de modelos incorrectos.

A pesar de que Aset-Solver ofrece algunos mecanismos de optimización, en la investigación realizada del sistema, se descubrió que aún se encuentra en una etapa de desarrollo y no es lo suficientemente robusto para representar problemas de optimización o al menos ninguno de los que se propusieron en este trabajo.
Los ejemplos incluidos en el sitio web del sistema son muy pequeños y sencillos, por lo que en nuestro interés de probar algo más significativo, se contactó al diseñador del sistema para transformar o incluso codificar otros problemas, pero nos sugirió usar CR-Prolog, en su opinión más adecuado y con más soporte de investigación por parte de ellos. El código en CR-Prolog del problema 3.6 se localiza en el Apéndice C.

Referente a la documentación, cada sistema ofrece manuales o tutoriales, pero lo mejores son los proporcionados por DLV que fueron claros, completos y útiles, ya que el usuario aprende el uso del sistema de una forma amigable y proporciona suficientes ejemplos para conocerlo. Aunque en el tutorial se detectó un ejemplo que generaba la salida incorrecta, este comportamiento no fue exclusivo de DLV, todos los sistemas mostraron inconsistencias en casos que ya estaban probados por ellos.

DLV permite de forma transparente la configuración de su ambiente en diferentes plataformas; lo anterior se probó en Unix y Windows; para A-POL por estar implementado en Java, forzosamente necesita la versión correcta de JRE, además de esto, su instalación es sencilla, contrariamente a Smodels y Aset-solver cuya configuración debió efectuarla un experto en el ambiente operativo.