RESUMEN.

La Programación con Restricciones, en ingles Constraint Programming (CP), es un paradigma de programación, donde las relaciones entre las variables son expresadas en términos de restricciones. Actualmente es usada como una tecnología para la descripción y resolución de problemas combinatoriales particularmente difíciles, especialmente en las áreas de planificación y programación de tareas. Dentro del estudio de la Programación con Restricciones encontramos una serie de Lenguajes de Programación, que permiten integrar restricciones de manera de dar soluciones a diferentes problemas. Es por este motivo que se propone un marco de comparación para evaluar estos lenguajes, proporcionando así una guía para su selección. Existen dos grandes paradigmas en los cuales nos vamos a centrar, el Orientado a Objetos y el Lógico. Hay dos razones para adoptar la tecnología de Lenguajes de Programación Lógicos para solucionar un problema. La primera es su expresividad permitiendo una solución declarativa con un código legible que es vital para el mantenimiento y el segundo es la disposición para una implementación eficiente para procedimientos de cómputo costosos. Mientras tanto los Lenguajes Orientado a Objetos son de implementación más sencilla y una vez que tienen el apoyo de clases específicas para la Programación con Restricciones se vuelven fácilmente adaptables. Una vez propuesto un mecanismo general de comparación parametrizable, se realiza la comparación entre Eclipse, por parte de los Lenguajes Lógicos y en GeCode por parte de los Lenguajes Orientados a Objetos.

PALABRAS CLAVES: Programación con Restricciones, Lenguajes de Programación Lógicos, Lenguaje de Programación con Restricciones, Lenguaje de Programación Orientado a Objetos, Eclipse, GeCode.

ABSTRACT

Constraint Programming (CP) is a programming paradigm where relationships among variables are expressed in terms of constraints. Currently, it is used as a technology to describe and solve particularly difficult combinatorial problems, especially in the areas of task planning and programming. Within the study of Constraint Programming we find a series of Programming Languages, which allow integrating constraints in order to solve different problems. For this reason, a comparison framework is proposed to evaluate these languages, providing a guide to select them. We will focus on two leading paradigms; Object-Oriented and Logic. There are two reasons to adopt the Logic Programming Language technology in order to solve a problem. Firstly, its expressiveness, permitting a declarative solution with a readable code –vital for maintenance– and secondly, the layout for an efficient implementation for costly computation procedures. Meanwhile, Object-Oriented Languages are simpler to implement, and once they are supported by specific classes for Constraint Programming they become easy to adapt. Once a general parameterizable comparison mechanism is proposed, a comparison is performed between Eclipse by Logic Languages and GeCode by Object-Oriented Languages.

KEYWORDS: Constraint Programming, Logic Programming Languages, Constraint Programming Language, Object-Oriented Programming Language, Eclipse, GeCode.

Índice General

1 Introducción. 7

1.1 Motivación *9*

1.2 Objetivos *10*

1.2.1 Objetivo General 10

1.2.2 Objetivos Específicos 10

2 Programación con Restricciones. 11

2.1 Problemas de Satisfacción de Restricciones. *13*

2.1.1 Definiciones previas. 13

2.1.2 Complejidad. 15

2.1.2.1 Complejidad algorítmica. 15

2.1.2.2 Complejidad de un problema. 16

2.2 Técnicas de Consistencia. *17*

2.2.1 Consistencia de nodos. 17

2.2.2 Consistencia de arcos. 19

2.2.3 Consistencia de caminos. 21

2.3 Técnicas de Búsqueda. *22*

2.4 Técnicas Híbridas. *23*

2.5 Estrategias de Enumeración. *25*

2.5.1 Heurísticas de Selección de Variable. 25

2.5.2 Heurísticas de Selección de Valor. 26

3 Lenguajes de Programación con Restricciones. 28

3.1 Programación Lógica con Restricciones (CLP). *28*

3.1.1 Programación Lógica. 28

3.1.2 El esquema CLP. 30

3.1.3 Aplicaciones de CLP. 31

3.1.4 Lenguajes de Programación con Restricciones Lógicos. 31

3.1.4.1 Glass Box vs Black Box. 31

3.1.4.2 Lenguajes Glass Box. 32

3.1.4.3 Lenguajes Black Box. 33

3.2 Programación Orientada a Objeto con Restricciones. *35*

3.2.1 Programación Orientada a Objetos. 35

3.2.2 Lenguajes de programación con restricciones Orientados a Objetos. 37

4 Estudio comparativo. 40

4.1 Definición de criterios. *42*

4.1.1 Criterios Internos. 42

4.1.1.1 Expresividad. 43

4.1.1.2 Eficiencia. 45

4.1.1.3 Desarrollo de restricciones. 47

4.1.2 Criterios Externos. 50

4.2 Puntuación de criterios. *52*

4.2.1 Criterios Internos. 53

4.2.1.1 Expresividad 53

4.2.1.2 Eficiencia. 55

4.2.1.3 Desarrollo de restricciones 58

4.2.2 Criterios Externos. 59

4.3 Ponderación de criterios. *61*

5 Prueba de la Plantilla de Comparación. 63

5.1 Selección de Lenguajes para la comparación. *63*

5.2 Comparación de criterios bibliográficos para el Caso: Eclipse y GeCode/J......................................*64*

5.3 Comparación de criterios prácticos para el Caso: Eclipse y GeCode/J. *75*

5.3.1 Justificación del uso de Modelos. 75

5.3.2 Implementación de Modelos. 83

5.3.2.1 Modelo de N-reinas: 83

5.3.2.2 Modelo de Cuadrado Mágico: 84

5.3.3 Comparación práctica de los lenguajes usando Modelos. 85

Hardware y compiladores 86

Líneas de Código y reutilización 86

Uso de Disco 87

Uso de Memoria 87

Tiempo de Ejecución 88

Número de Backtracks 89

5.4 Evaluación y Ponderación final del Caso: Eclipse y GeCode/J. *90*

Conclusiones. 93

Bibliografía. 95

Índice de Figuras

Figura 2.1: Ejemplo de un CSP sobre dominios discretos..14

Figura 2.2: Ejemplo de No consistencia de nodo 18

Figura 2.3: Ejemplo de Consistencia de nodo 18

Figura 2.4: CSP No arco consistente 19

Figura 2.6: CSP Arco consistente 20

Figura 2.7: CSP No camino consistente...21

Figura 2.8: CSP Camino consistente 22

Figura 3.1: Niveles de abstracción de GeCode 38

Figura 4.1: Etapas de resolución de CSP...41

Figura 4.2: Características deseables en las etapas de modelado y resolución 42

Figura 5.1: Estructura global de aplicación de sistemas desarrollado por Parc Technologies para Eclipse......73

Figura 5.2: Solución para 4 reinas..78

Figura 5.3: Cuadrado Mágico n=3, número mágico 15...78

Figura 5.4: Modelo del problema de coloreo de mapa 79

Figura 5.5: Ejemplo de Sudoku 9x9 79

Figura 5.6: Tablero de solitario de 33 hoyos..82

Índice de Tablas

Tabla 4.1: Criterios internos de comparación..50

Tabla 4.2: Criterios de evaluación Externos 52

Tabla 4.3: Puntaje para tipificación y comprobación de tipos..54

Tabla 4.4: Fragmento de la tabla de puntuación y ponderación..61

Tabla 4.5: Ponderación de los grupos de evaluación..62

Tabla 5.1: Estrategias de selección de variables entera..70

Tabla 5.2: Estrategias para selección de valor 72

Tabla 5.3: Líneas de código utilizadas en cada lenguaje 86

Tabla 5.4: Cantidad de disco utilizado en cada implementación 87

Tabla 5.5: Memoria utilizada en la ejecución de cada instancia expresada en KB 87

Tabla 5.6: Tiempo de ejecución para N-reinas..88

Tabla 5.7: Tiempo de ejecución para Cuadrado Mágico. 88

Tabla 5.8: Número de Backtracks para encontrar la primera solución en N-reinas..89

Tabla 5.9: Ponderación de cada grupo de criterios para el Caso: Eclipse y GeCode/J 90

Tabla 5.10: Puntuación y ponderación de cada criterio en el Caso: Eclipse y GeCode/J. 91

Tabla 5.11: Puntuación y ponderación de grupos de criterios. 92

Índice de Algoritmos

Algoritmo 1: Backtracking Cronológico 24

Algoritmo 2: Pseudo código de Forward Checking..25

CAPÍTULO 1

Introducción

1 Introducción.

La Programación con Restricciones (en ingles Constraint Programming, CP), ha sido utilizada de manera efectiva para resolver grandes y complejos problemas combinatoriales [1], [2], estos problemas se pueden modelar como Problemas de Satisfacción de Restricciones (en ingles Constraint Satisfaction Problem, CSP), [3], [4], el cual es definido mediante un conjunto de variables, cada una con un dominio asociado, más un conjunto de restricciones. Los dominios corresponden a un conjunto de posibles valores que se pueden asignar a las variables, en algunos casos los dominios se pueden definir de manera booleana. Cada restricción es definida sobre un conjunto de variables restringiendo la combinación de valores que pueden ser asignados a esas variables. Resolver un CSP consiste en encontrar una asignación de valores a las variables de manera tal que se satisfagan todas las restricciones [5]. Para la resolución de estos problemas, se utiliza una aproximación completa que consiste en alternar fases de propagación de restricciones y enumeración [3], [5]. Estos conceptos serán explicados más adelante, ya que son cruciales en el rendimiento del proceso de resolución de un problema con CP.

Existe una clara relación de la metodología de CSP con la Programación con Restricciones donde las relaciones entre las variables también pueden establecerse en forma de restricciones embebidas en un lenguaje de programación y, particularmente, con la Programación Lógica. De esta forma, la Programación Lógica con Restricciones (en ingles Constraint Logic Programming, CLP) puede verse como un CSP donde las restricciones se reescriben como predicados y la unificación en el lenguaje de programación se sustituye por la resolución por coerción en un dominio específico (satisfacción de restricciones) [6].

La programación con restricciones lógicas ha tenido un gran éxito en la resolución de CSP. La evidencia del éxito de CLP ([7], [8], [9]) se puede representar gracias al aumento de los sistemas de CLP que ahora son utilizados para muchos usos de la vida real [10]. Hay dos razones principales de este éxito: primero, CLP amplía el paradigma de programación lógica permitiendo ser más declarativo y las soluciones mas legibles, y en segundo lugar, apoya la propagación de las restricciones para los dominios específicos, proporcionando una implementación eficiente para procedimientos de cómputo costosos.

Sin embargo, los sistemas de CLP se diferencian perceptiblemente en cómo las soluciones se pueden expresar y la eficacia de su ejecución. Es importante que estos dos factores sean considerados al elegir el mejor sistema de CLP para un uso particular. De hecho, la selección incorrecta de un sistema CLP puede ser desastrosa, no solamente por la eficiencia en su funcionamiento, sino también con respecto a la claridad del código de la solución, importante para las modificaciones futuras. Por otra parte también se han utilizado lenguajes de programación que no son lógicos, y que se han vuelto tremendamente populares para resolver CSP, este es el caso de los Lenguajes Orientados a Objetos.

De los dominios, el dominio finito (en ingles finite domiain, FD) es uno de los mas estudiados, el cual proporciona un marco de trabajo adecuado para solucionar problemas discretos de satisfacción de restricciones. El FD es particularmente útil para modelar problemas tales como programación de tareas (en ingles scheduling), planeación, embalaje, generación de horarios (en ingles timetabling), por consiguiente la mayoría de los sistemas proporcionan bibliotecas substanciales de FD. En este trabajo solamente se consideran dominios finitos y casos de dominio booleano, los que pueden verse como un caso particular de dominio finito [6].

Se realiza un análisis de las características de lenguajes CLP (conocidos como solver CLP). En los que se destaca Eclipse [11], por parte de los Lenguajes Lógicos y GeCode [61] por parte de los Lenguajes Orientados a Objetos. Estos sistemas particulares cubren las clases principales de solvers de FD y son populares dentro de la comunidad de Programación con Restricciones.

Se elaborará una guía para la selección de Lenguajes de Programación con Restricciones. Definiendo una serie de criterios para realizar una comparación entre estos lenguajes, la cual sea un elemento parametrizable de acuerdo a los objetivos del evaluador para la selección de un determinado lenguaje. A modo de prueba de esta guía se realiza una instancia particular la cual podríamos llamar un *caso de evaluación*, la cual se realiza usando Eclipse y GeCode.

Entre los criterios de comparación hay criterios que se pueden evaluar de manera bibliografía y criterios que se deben evaluar de manera práctica. Para realizar las evaluaciones de criterios prácticos se debe usar un problema (a los que también podemos referirnos como modelos) para resolver. Por esto se realiza un estudio previo para justificar el uso de cada problema ó modelo, y de acuerdo a esto se seleccionan algunas opciones de modelos, como prueba original para la comparación entre diferentes sistemas.

1.1 Motivación.

En la literatura actual sobre los problemas de satisfacción con restricciones (Constraint Satisfaction Problem, CSP,) se abordan las técnicas clásicas empleadas para su resolución y, más particularmente, otras técnicas usadas sobre áreas específicas de aplicación (tales como la diagnosis, las bases de datos o la recuperación de información). Esta literatura cubre muchos de los aspectos relacionados con los CSPs pero obvia un área muy importante como es la integración de restricciones en los lenguajes declarativos (especialmente los lógicos). En realidad, la programación lógica con restricciones (Constraint Logic Programming, CLP,) [9,20] es uno de los campos de investigación más activos en la comunidad de las restricciones puesto que esta directamente relacionado con el modelado (a alto nivel) de soluciones a los problemas de satisfacción con restricciones; mas aun, uno no puede (ni debe) asumir que todas las restricciones estarán disponibles al comienzo del proceso de búsqueda de las soluciones y el usuario podría querer construir el CSP a resolver de forma incremental y “representar las restricciones a través de formulas pertenecientes a un cierto lenguaje de programación en vez de representar estas mediante conjuntos de tuplas de valores". Dado que el modelado de un CSP mediante programación lógica varia dependiendo del sistema CLP utilizado surge la motivación de incorporar un nuevo paradigma considerando ahora la programación Orientada a Objeto.

Como ya se dijo, la selección de un determinado lenguaje se vuelve una tarea difícil, dada la cantidad de solvers disponibles, y esta selección en algunos casos puede ser critica para la implementación y posterior solución de un CSP. Por este motivo se busca generar una guía para la selección de Lenguajes de Programación con Restricciones. Definiendo una serie de criterios para realizar una comparación entre estos lenguajes, la cual sea un elemento parametrizable de acuerdo a los objetivos del evaluador para la selección de un determinado lenguaje independiente del paradigma que estos cumplan.

Los sistemas CLP y Orientado a Objetos (OO) se diferencian en cómo las soluciones se pueden expresar y la eficacia de su ejecución. Es importante que estos dos factores sean considerados al elegir el mejor sistema para un uso particular. De hecho, la selección incorrecta de un sistema puede ser desastrosa, no solamente por la eficiencia en su funcionamiento, sino también con respecto a la claridad del código de la solución, importante para las modificaciones futuras.

Hoy en día la gama de problemas que se pueden resolver con programación con restricciones es amplia, varia desde problemas de ingeniería, finanzas, scheduling, timetabling, enrutamientos (en ingles routing), entre otros. Al momento de realizar pruebas no existe una justificación que indique el porque se utilizan determinados problemas. Es por eso que se complementa este informe con un pequeño estudio que justifique el uso de algún determinado problema de prueba, ya que solo se cuenta con la utilización de algunos problemas clásicos, pero sin ninguna información que guié el uso de uno u otro problema.

1.2 Objetivos.

A continuación se dará a conocer los objetivos generales y específicos que tiene el proyecto presentado en este documento.

1.2.1 Objetivo General.

Elaborar una guía para el estudio comparativo y selección de Lenguajes de Programación especializados en el modelado y resolución de Problemas de Satisfacción de Restricciones.

1.2.2 Objetivos Específicos.

En este punto se dará a conocer cuales son los objetivos específicos que tiene este proyecto, los cuales tienen directa relación con el objetivo general expuesto en el punto anterior. A continuación se presenta un punteo con los objetivos específicos de este proyecto:

- Estudiar la resolución de problemas mediante CSP.
- Estudiar y comprender el funcionamiento de los diferentes Lenguajes de Programación con Restricciones Lógicos y Orientado a Objetos, seleccionando uno de cada tipo.
- Definir una serie de criterios para comparar los lenguajes seleccionados.
- Generar una plantilla para la evaluación y ponderación de los criterios.
- Comparar dos lenguajes haciendo uso de los criterios definidos.
- Seleccionar e implementar un par de problemas para los lenguajes en evaluación.
- Realizar el análisis comparativo entre lenguajes según criterios y plantilla de comparación.

CAPÍTULO 2

Programación con Restricciones

2 Programación con Restricciones.

La Programación con Restricciones involucra tanto el modelamiento como la resolución de sistemas basados en restricciones. Como disciplina abarca áreas del conocimiento y la investigación como las matemáticas discretas, el análisis de intervalos, la programación matemática, la inteligencia artificial y el cálculo formal.

Entre sus principales campos de aplicación se encuentran los problemas de optimización combinatorial y ordenamiento, análisis financiero, diseño y construcción de circuitos integrados [23], la biología molecular [24] y la resolución de problemas geométricos [25], entre otras. Como idea general, la programación con restricciones intenta diseñar estrategias de resolución eficientes, usando el conocimiento y la estructura del problema.

Uno de los primeros indicios de la programación con restricciones se encuentra en el año 1947, cuando George Dantzig crea el método simplex para programación lineal, descrito por primera vez en su paper *Programming in a linear structure* [26] (Programación en una estructura lineal). El termino *programming* (programación) estaba referido a los tipos de problemas abordados por Dantzig en aquellos años, denominados *programming problems* (problemas de programación), relacionados con la investigación en *devise programs of activities for future conflicts* del departamento de defensa de los Estados Unidos. Posteriormente se utiliza el término programación lineal en lugar de programación en una estructura lineal, y los problemas pertenecientes a esta área reciben el nombre de problemas de programación lineal. De esta forma se asocia el término programación con un tipo de problema matemático específico en la literatura de la Investigación de Operaciones.

La programación con restricciones, comienza a desarrollarse a mediados de los años 80, como una técnica de la informática que combina el desarrollo de la comunidad de Inteligencia Artificial con los avances en lenguajes de programación. En este contexto, la palabra programming se refiere específicamente a un programa computacional. De esta forma, la programación con restricciones es en sí, una técnica de programación de computadores desde un punto de vista lógico. La programación lógica se caracteriza por ser declarativa y utilizar un estilo relacional basado en la lógica de primer orden.

Si bien en sus comienzos se utilizaban algoritmos simples para resolver las estructuras lógicas de un problema, el avance en investigación de nuevas técnicas y algoritmos más poderosos han extendido enormemente sus potencialidades.

Desde el punto de vista de la arquitectura, la programación con restricciones posee dos niveles: el componente restringido y el componente de programación. El primer nivel permite formular la definición del problema desde el punto de vista de sus variables y restricciones, de una forma relativamente simple y sin necesidad de grandes conocimientos de lenguajes de programación. Por otro lado, el componente de programación permite escribir algoritmos específicos para indicar de qué manera deberán ser modificadas las variables para encontrar valores que satisfagan las restricciones.

Para facilitar la aplicación de este tipo de técnicas a problemas complejos se han desarrollado diversos lenguajes de programación, que van desde ALICE [27] en la década del 70 hasta los mas actuales, como OPL o ILog [28],[29], a fines de la década del 90. Una descripción mas detallada de los inicios de la programación con restricciones y su evolución, puede ser obtenida en [30].

La programación de restricciones puede dividirse en dos ramas claramente diferenciadas: la "satisfacción de restricciones" y la "resolución de restricciones". Ambas comparten la misma terminología, pero sus orígenes y técnicas de resolución son diferentes. La satisfacción de restricciones trata con problemas que tienen dominios finitos, mientras que la resolución de restricciones está orientada principalmente a problemas sobre dominios infinitos o dominios más complejos. En este capítulo, se tratarán principalmente los problemas de satisfacción de restricciones (CSP). Los conceptos clave en esta metodología corresponden a los aspectos de:

- La modelización del problema, que permite representar un problema mediante un conjunto finito de variables, un dominio de valores finito para cada variable y un conjunto de restricciones que acotan las combinaciones válidas de valores que las variables pueden tomar. En la modelización CSP, es fundamental la capacidad expresiva, a fin de poder captar todos los aspectos significativos del problema a modelar.
- Técnicas inferenciales que permiten deducir nueva información sobre el problema a partir de la explícitamente representada. Estas técnicas también permiten acotar y hacer más eficiente el proceso de búsqueda de soluciones.
- Técnicas de búsqueda de la solución, apoyadas generalmente por criterios heurísticos, bien dependientes o independientes del dominio. El objetivo es encontrar un valor para cada variable del problema de manera que se satisfagan todas las restricciones del problema. En general, la obtención de soluciones en un CSP es NP-completo, mientras que la obtención de soluciones optimizadas es NP-duro, no existiendo forma de verificar la optimalidad de la solución en tiempo polinomial. Por ello, se requiere una gran eficiencia en los procesos de búsqueda.

2.1 Problemas de Satisfacción de Restricciones.

En este capítulo se examinan las principales definiciones relacionadas con los Problemas de Satisfacción de Restricciones. Además, se presentan ejemplos de CSP sobre dominios discretos y continuos. Finalmente se revisan las nociones de complejidad, asociadas tanto a las características del problema en sí, como al algoritmo de búsqueda de solución. Esto para entender la complejidad de un problema de satisfacción de restricciones.

Los CSP tienen una gran importancia en diversas áreas del que hacer humano. Muchos de los problemas de la vida cotidiana pueden ser modelados a través de un conjunto de entidades denominadas variables, a las cuales se asocian conjuntos de valores posibles denominados dominios y que poseen conjuntos de relaciones entre sí, las cuales reciben el nombre de restricciones. En este contexto un modelo es, entonces, una representación simplificada de la realidad a través de estas entidades.

2.1.1 Definiciones previas.

Dada la gran cantidad de terminologías existentes para denotar un CSP, es necesario formalizar su definición y la de sus componentes relacionados. En este trabajo, se tratara un subconjunto de los problemas de satisfacción de restricciones, que corresponden a los CSP con dominios continuos que involucran ecuaciones de distancia. La definición formal de un CSP se presenta a continuación:

CSP: Un problema de satisfacción de restricciones $P = (X, D, C)$ es una tripleta con $X = \{x_1,\ldots, x_n\}$ conjunto de variables, $D = \{D_{x1},\ldots, D_{xn}\}$ conjunto de dominios asociados a las variables y $C = \{c_1,\ldots, c_m\}$ conjunto de restricciones sobre las variables.

Espacio de búsqueda: Sea $P = (X, D, C)$ un problema de satisfacción de restricciones con $D = \{D_{x1}, \ldots, D_{xn}\}$. El espacio de búsqueda E asociado a P se define como $E = D_{x1} \times\ldots \times D_{xn}$ el producto cartesiano de los dominios asociados a cada una de las variables del problema.

Solución: Una solución $\vec{s} = (a_1,\ldots, a_n)$ de un CSP $P = (X, D, C)$ es una asignación de valores a las variables donde $x_1 = a_1 \in D_{x1},\ldots, x_n = a_n \in D_{xn}$, tal que $\forall\ c \in C, c(\vec{s})$ es verdadera.

Espacio solución: Sea $P = (X, D, C)$ un problema de satisfacción de restricciones con espacio de búsqueda E. Sea $S \in E$. Se dice que S es el espacio solución de P si $\forall\ \vec{s} \in S$, $\vec{s}$ es solución del problema y $\forall\ \vec{e} \in E - S$, $\vec{e}$ no es solución del problema.

En las definiciones previas no se ha restringido el tipo de dominios asignados a las variables ni las características de las restricciones. Si bien el dominio de las variables depende del problema en sí, es posible clasificarlo en dos grandes grupos: dominios discretos y dominios continuos. A continuación se ejemplifican los dos grupos, el primero corresponde a un problema clásico de satisfacción de restricciones con dominios discretos denominado coloreo de grafos, mientras que el segundo corresponde a un problema con dominios continuos de ecuaciones de distancia.

1. Problema de coloreo de grafos con 3 colores: El problema de la figura 2.1 consiste en un grafo de tres nodos: X, Y , Z. Para cada nodo se debe seleccionar un color desde su dominio, de tal forma que dos nodos conectados por un arco no posean igual color. La solución al problema consiste en determinar una asignación simultánea de colores para todos los nodos del grafo, asegurando que todo par de nodos adyacentes posea diferente color. Por ejemplo, la asignación Nodo(X)=negro, Nodo(Y)=azul y Nodo(Z)=rojo, es una solución al problema.

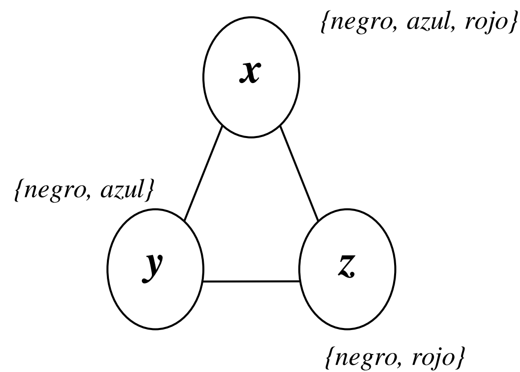

Figura 2.1: Ejemplo de un CSP sobre dominios discretos

Dado que el dominio de las variables es discreto, se dice que el CSP es discreto, y más específicamente, un CSP sobre dominios discretos. Una aplicación directa de este tipo de problemas se encuentra en el diseño de mapas. El objetivo es colorear un mapa respetando un conjunto máximo de colores, y asegurando que dos regiones colindantes del mapa no posean igual color. Formalmente el problema consiste en resolver $P = (X, D, C)$ con $X = \{x, y, z\}$, $D = \{\{negro, azul, rojo\}, \{negro, azul\}, \{negro, rojo\}\}$, y $C = \{\{x \neq y\}, \{x \neq z\}, \{y \neq z\}\}$.

La versatilidad de este problema permite usarlo y aplicarlo a problemas de asignación horaria y planificación, entre otros.

2. Problema de ecuaciones de distancia: El problema consiste en dos puntos en el plano con ubicación fija X e Y, y un tercer punto Z que debe ser posicionado a una distancia no mayor a r_1 del punto X y r_2 del punto Y. Formalmente se trata de resolver $P = (X, D, C)$ con $X = \{ \vec{x}, \vec{y}, \vec{z} \}$, $D = \{D_x = D_y = D_z = [0, 100] \times [0, 100]\}$, y $C = \{\{ \vec{x} = \vec{a} \}, \{ \vec{y} = \vec{b} \}, \{d(\vec{z}, \vec{x}) \cdot r_1\}, \{d(\vec{z}, \vec{y}) \cdot r_2\}\}$. La solución a este problema consiste en determinar la zona del plano que cumple con todas las restricciones simultáneamente.

   La zona demarcada representa todos aquellos puntos del plano que cumplen con las restricciones planteadas. Dado que el dominio de las variables (punto Z) es continuo, existen infinitas soluciones para el problema. Este tipo de problema recibe el nombre de CSP numérico o CSP sobre dominios continuos. Una aplicación directa de este tipo de problemas se encuentra en la determinación de zonas de interferencia entre dos torres emisoras de ondas con cobertura limitada. En este caso interesa determinar las regiones del espacio que podrían presentar problemas de interferencia, debido a la influencia de diferentes ondas emanadas de artefactos emisores fijos.

Existe además una tercera clasificación de problemas: los CSP mixtos, que son aquellos que involucran variables con dominios discretos y continuos. Esta clase de CSP se denomina comúnmente CSP híbridos, y poseen un grado de complejidad adicional, ya que requieren una combinación de técnicas diseñadas para cada tipo de dominio. Un listado general de ejemplos de CSP y técnicas aplicadas para su resolución se pueden consultar en [21].

2.1.2 Complejidad.

El tema de la complejidad puede verse de dos formas distintas: algoritmo de búsqueda de solución y problema en sí. La principal diferencia entre ambas es el centro de estudio. Mientras la complejidad del algoritmo estudia las características propias de las técnicas para resolver un problema, la complejidad del problema estudia las características del problema en sí. Ambas alternativas poseen gran importancia en el estudio y solución de problemas complejos y pueden ser complementadas. La primera sirve para comparar dos algoritmos que resuelven el mismo problema y la segunda permite clasificar problemas basados en la dificultad de su resolución.

2.1.2.1 Complejidad algorítmica.

Frecuentemente es necesario comparar algoritmos para medir cual de ellos posee el mejor desempeño. Existen básicamente dos formas de realizar esta comparación: la evaluación comparativa y el análisis de algoritmo.

Evaluación comparativa:

Consiste en ejecutar ambos programas en el mismo entorno (misma maquina) con un sistema operativo, un compilador y un conjunto de datos de entrada en particular y medir cual de ellos se ejecuta en forma mas rápida o cual de ellos necesita menos memoria para su ejecución. El principal problema de este tipo de comparación es la particularidad de la medición. Por esta razón, este método de comparación es poco utilizado en la práctica, y solo esta limitado a sistemas muy particulares en entornos muy estáticos**.**

Análisis de algoritmo:

Este método es independiente de las entradas y del tipo de implantación realizada. Consiste básicamente en realizar un análisis de la cantidad de tareas (instrucciones) que realizará cada algoritmo abstrayendo las entradas y su implantación. El análisis se realiza en dos pasos:

1. **Abstrayendo la entrada**. Es decir, caracterizando al problema a través de un parámetro en particular. Una posible medida es la longitud de la secuencia (cantidad de variables o valores de entrada) que se caracteriza con un número n.

2. **Abstrayendo la implantación**. Es decir, encontrando una medida de desempeño que no dependa de la maquina, sistema operativo o compilador utilizado. Una posible medida es la cantidad de líneas de código ejecutadas, o la sumatoria de las instrucciones básicas realizadas (sumas, restas, asignaciones, comparaciones, entre otras). De esta forma se obtiene una caracterización, basada en los datos de entrada, denominada $T(n)$.

2.1.2.2 Complejidad de un problema.

De igual forma que se realiza el análisis de algoritmos para determinar su complejidad, se estudian las características de los problemas que los hacen más o menos difíciles de resolver. La principal diferencia global entre los problemas es el tiempo requerido para resolverlos. En primer lugar se encuentran aquellos que son susceptibles de ser resueltos en un tiempo polinomial. A ellos se les clasifica comúnmente como problemas P. Por otra parte, existen aquellos problemas para los cuales no existe aun un algoritmo capaz de resolverlos en este tiempo (para cualquier instancia particular del problema), por lo que se los clasifica como problemas NP. Los problemas pertenecientes a la clasificación P son en general sencillos, ya que suelen ser resueltos por algoritmos del tipo $O(f(n))$ donde $f(n)$ representa un polinomio particular. A esto problemas se les denomina genéricamente manejables. Por otra parte, los problemas clasificados como NP se caracterizan porque el tiempo necesario para resolverlos crece, al menos, exponencialmente ante un crecimiento lineal en el tamaño de la entrada. Esto dificulta su resolución, incluso para entradas de tamaño reducido. Una sub-clasificación de problemas NP es la denominada NP completos. Este tipo

de problemas corresponde a los mas difíciles de resolver de la clase NP. Se ha demostrado que si es posible resolver uno de estos problemas en tiempo polinomial (para cualquier instancia particular), entonces todos ellos pueden ser resueltos en tiempo polinomial. Una explicación mas detallada acerca de complejidad e intratabilidad puede ser obtenida en [22].

2.2 Técnicas de Consistencia.

Las técnicas de consistencia juegan un papel fundamental en la aplicación de algoritmos de propagación para reducir los dominios de las variables pertenecientes a un problema. De manera general, las técnicas de consistencia examinan el dominio de las variables en busca de valores o conjunto de valores que no podrán ser asignados respetando las restricciones del problema. Se denominan técnicas de filtrado y se aplican para encontrar un CSP equivalente (si es que ambos tienen el mismo espacio de solución).

Una de las técnicas de consistencia más difundida y utilizada para reducir dominios en CSP con dominios discretos en la arco-consistencia [31]. Una de las principales dificultades que suelen encontrarse en los algoritmos de búsqueda es la aparición de inconsistencias locales. Las inconsistencias locales son valores individuales (o combinación de valores) de las variables que no pueden participar en la solución porque no satisfacen alguna propiedad de consistencia, es decir, no satisfacen alguna restricción. En la literatura pueden encontrarse diferentes niveles de consistencia local [1], [2], [3], [32].

2.2.1 Consistencia de nodos.

Forzar este nivel de consistencia asegura que todos los valores en el dominio de una variable satisfacen todas las restricciones unárias sobre esa variable [33], [1], [32].

$$\forall x_i \in \mathrm{X}, \forall a \in D_{x_i} : a \text{ satisface } C_{x_i}$$

Ejemplo: Considere el CSP constituido por el conjunto de variables $X = x_1, x_2$, sus dominios $D_{x1} = \{1, 2, 3, 4, 5\}$, $D_{x2} = \{1, 2, 3, 4, 5\}$ respectivamente, y el conjunto de restricciones $C = x_1 \leq 3, x_2 \geq 1, x_1 \neq x_2$

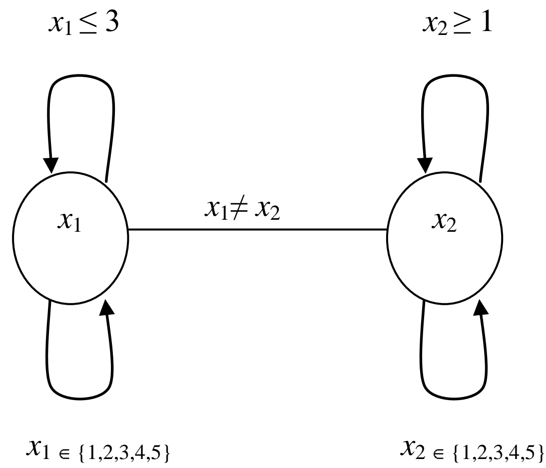

Figura 2.2: Ejemplo de No consistencia de nodo

Es posible apreciar en la figura 2.2 que el CSP planteado no es nodo consistente, esto debido a que el nodo x_1 no satisface la consistencia de nodo, es decir, x_1 posee valores en su dominio que no satisfacen la restricción unaria $x_1 \leq 3$. De esta manera, para que el CSP anterior sea nodo consistente, solo bastaría con eliminar los valores del dominio de la variable x_1 que no cumplen la restricción unaria (4 y 5), quedando el grafo de la figura anterior de la siguiente forma:

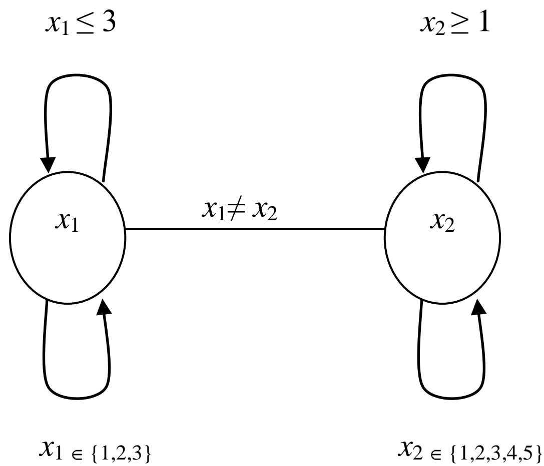

Figura 2.3: Ejemplo de Consistencia de nodo

2.2.2 Consistencia de arcos.

Se dice que un CSP es arco consistente si toda restricción binaria es arco consistente [3]. Una restricción binaria C sobre las variables x_1 y x_2, cuyos dominios son D_{x1} y D_{x2} respectivamente, es arco consistente si:

$$\forall a \in D_{x_1}\ \exists b \in D_{x_2} : (a,b) \quad \text{satisface} \quad C$$

$$\forall b \in D_{x2}\ \exists a \in D_{x_1} : (a,b) \quad \text{satisface} \quad C$$

De esta manera, una restricción binaria es arco consistente si cada valor en cada dominio tiene un soporte en el otro dominio, donde b es llamado un soporte para a si el par (a, b) satisface la restricción [3].

Un caso particular de la arco consistencia es el arco consistencia dirigida [3][1], donde dado un orden linear→ sobre las variables consideradas, se requiere la existencia de soportes solo en una dirección, esto es:

Dadas las condiciones:

$$\forall a \in D_{x_1}\ \exists b \in D_{x_2} : (a,b) \quad \text{satisface} \quad C, \text{ proporcionado el orden } x_1 \rightarrow x_2$$

$$\forall b \in D_{x2}\ \exists a \in D_{x1} : (a,b) \quad \text{satisface} \quad C, \text{ proporcionado el orden } x_2 \rightarrow x_1$$

Solo una de ellas necesita ser chequeada.

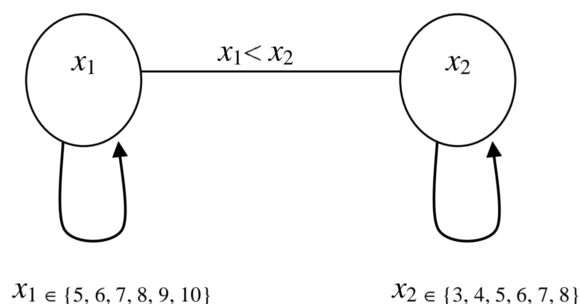

Figura 2.4: CSP No arco consistente

Ejemplo: El CSP mostrado en la Figura 2.4, el cual esta constituido por $X = \{x_1, x_2\}$, $D_{x1} = \{5, 6, 7, 8, 9, 10\}$, $D_{x2} = \{3, 4, 5, 6, 7, 8\}$ y $C = \{ x_1 < x_2\}$, no es arco consistente porque si se toma el valor 8 en el dominio de x_1, no existe un valor en el dominio de x_2 tal que se satisfaga la restricción $x_1 < x_2$. Por otra parte, el mismo CSP no es

direccionalmente arco consistente, ya que si se considera el orden $x_1 \rightarrow x_2$ con $x_1 = 8$, no existe un valor en D_{x2} que satisfaga la restricción binaria del CSP. Por su parte, si el orden considerado es $x_2 \rightarrow x_1$ con $x_2 = 4$, no hay un valor en D_{x1} que satisfaga la restricción mencionada.

Para que el CSP anterior sea direccionalmente arco consistente bajo el orden $x_2 \rightarrow x_1$, es preciso ajustar los dominios de manera tal que cada elemento en el dominio de x_2 tenga un soporte en D_{x1}, este ajuste es mostrado en la Figura 2.5, donde además es posible apreciar que el CSP aún no es arco consistente. Para cambiar esto ultimo, y alcanzar el arco consistencia los dominios de x_1 y x_2 deben ser reducidos tal como muestra la Figura 2.6.

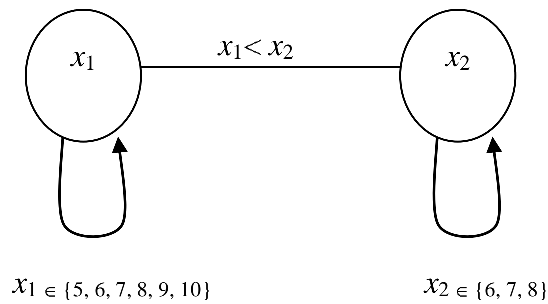

Figura 2.5: CSP Directamente Arco consistente bajo el orden $x_2 \rightarrow x_1$

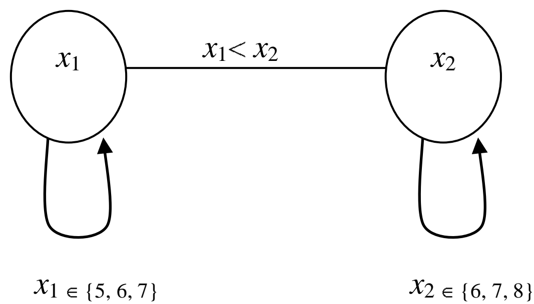

Figura 2.6: CSP Arco consistente

2.2.3 Consistencia de caminos.

Requiere que para cada par de valores a y b de dos variables x_i y x_j respectivamente, las asignaciones (x_i, a) y (x_j, b) satisfagan la restricción directa entre x_i y x_j, y que además exista un valor para cada variable a lo largo del camino entre ellas de forma que todas las restricciones a lo largo del camino se satisfagan [33] [1]. Si cada camino de largo 2 en un grafo de restricciones cumple con la consistencia de camino, entonces el grafo de restricciones es camino consistente a nivel global [34].

Ejemplo: El CSP mostrado en la Figura 2.7, el cual esta constituido por $X = \{ x_1, x_2, x_3\}$, $D_{x1} = \{4, 5\}$, $D_{x2} = \{3, 4\}$, $D_{x3} = \{2, 3\}$ y $C = \{ x_1 > x_2, x_2 > x_3, x_1, x_3\}$, no camino consistente porque si se toma el valor 4 en el dominio de x_1, y el valor 3 en el dominio de x_3 no existe un valor en el dominio de x_2 tal que se satisfaga la restricción $x_2 < x_3$. De esta manera es preciso ajustar los dominios, quedando el grafo de restricciones de la forma mostrada en la Figura 2.8.

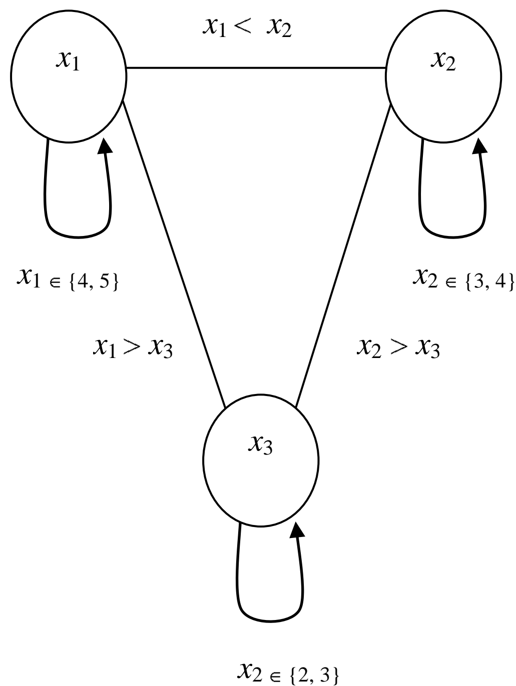

Figura 2.7: CSP No camino consistente

2.3 Técnicas de Búsqueda.

Este tipo de técnicas se centran en explorar el espacio de búsqueda del problema a resolver. Estas técnicas pueden ser completas, las cuales exploran todo el espacio en busca de una solución, o incompletas si solamente exploran una parte del espacio de búsqueda. Las técnicas que exploran todo el espacio, garantizan encontrar una solución, si existe, o demuestran que el problema no tiene solución. Las técnicas completas más usadas son:

Generación y Test (GT): de forma sistemática genera todas las posibles asignaciones completas [32] [33]. Cuando finaliza de generar todas las asignaciones completas, comprueba si alguna de ellas es una solución, es decir, comprueba si alguna de las asignaciones satisface todas las restricciones, donde la primera que satisfaga todas las restricciones será la solución del problema.

Backtracking Cronológico (BT): recorre el árbol utilizando búsqueda primero en profundidad, y en cada nueva instanciación comprueba si las instanciaciones parciales ya realizadas son localmente consistentes. Si es así, continua con la instanciación de una nueva variable. Por el contrario, si detecta inconsistencia, intenta asignar un nuevo valor a la última variable instanciada, si es posible, y en caso contrario retrocede a la variable asignada inmediatamente anterior [32, 33].

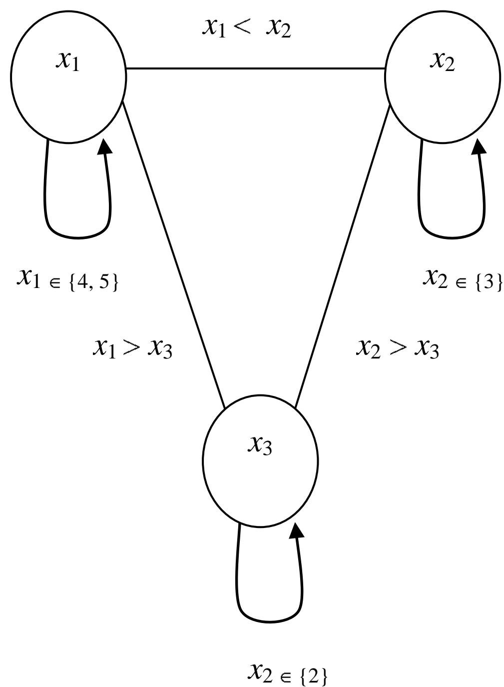

Figura 2.8: CSP Camino consistente

2.4 Técnicas Híbridas.

Las técnicas de búsqueda y algunas de las técnicas de consistencia pueden utilizarse independientemente para solucionar totalmente un problema de satisfacción de restricciones, pero esto rara vez sucede, por lo cual una combinación de ambos acercamientos es la manera más común para solucionar un CSP. Al incluir las técnicas de consistencia en el proceso de búsqueda, se obtienen técnicas híbridas de resolución donde la consistencia permitirá acotar el espacio de búsqueda, reduciendo de esta manera el costo de dicho proceso.

Algoritmos Look-Backward [1] Estos algoritmos tratan de explotar la información del problema de manera de tener un mejor comportamiento en aquellas situaciones sin salida. Los algoritmos Look-Backward realizan comprobación de consistencia hacia atrás, es decir, entre la variable actualmente instanciada y las pasadas ya instanciadas.

A continuación se muestran algunas de las variantes de algoritmos Look-Backward:

Backjumping (BJ): en lugar de retroceder a la variable anteriormente instanciada, tal como lo hace el Backtracking Cronológico, BJ salta a la variable x_j que es más cercana a la variable actual x_i, donde $j < i$, y con la cual esta en conflicto [35, 1].

Conflict-directed Backjumping (CBJ): en este algoritmo, cada variable x_i tiene un conjunto conflicto formado por las variables pasadas que están en conflicto con x_i. De esta manera, al comprobar que la consistencia entre la variable actual x_i y la variable pasada x_j falla, x_j se agrega al conjunto conflicto. En situaciones sin salida, CBJ salta a la variables más profunda en su conjunto conflicto, por ejemplo x_k con $k < i$, y al mismo tiempo el conjunto conflicto de x_i es agregado al conjunto conflicto de x_k con objeto de no perder información [35, 1].

```
PROCEDIMIENTO Backtracking (k, V[n]):
INICIO :

1 V[n] = Selección(dk);
2 if Comprobar (k, V[n]) then
3       if k = n then
4             Devolver V[n];
5       else
6             Backtracking (k+1, V[n]);
7       end
8 else
9       if quedanvalores (dk) then
10            Backtracking (k, V[n]);
11      else
12            if k = 1 then
13                  Devolver 0;
14            else
15                  Backtracking (k-1, V[n]);
16            end
17      end
18end
```

Algoritmo 1: Backtracking Cronológico

Algoritmos Look-Ahead[4] : Estos algoritmos realizan comprobación de consistencia hacia adelante en cada instanciación, integrando un proceso inferencial durante el propio proceso de búsqueda. Esto se conoce como Propagación, lo cual permite: (i) acotar las restricciones y dominios de las variables futuras a instanciar, limitando el espacio de búsqueda, y (ii) encontrar las inconsistencias antes de que aparezcan. En síntesis, intentan descubrir si la actual asignación parcial podrá ser extendida a una solución global, provocando en caso contrario un punto de backtracking.

Forward Checking (FC): en cada paso de la búsqueda se comprueba hacia adelante la asignación de la variable actual con todos los valores de las variables futuras que están restringidas con la variable actual. Aquellos valores inconsistentes con la asignación actual son eliminados de los dominios, si luego de esta eliminación el dominio de la variable futura queda vacío, la instanciación de la variable actual se deshace y se prueba con un nuevo valor. Si ningún valor es consistente se lleva a cabo Backtracking Cronológico [35] [1] [3].

Minimal Forward Checking (MFC): en lugar de comprobar hacia adelante la asignación actual contra todos valores de las variables futuras, MFC solo comprueba con los valores de las variables futuras hasta que se encuentra un valor que es consistente. De esta manera, si el algoritmo retrocede solo realizará comprobaciones con los valores restantes aún no comprobados. [1].

1 Seleccionar x_i;

2 Instanciar $x_i \leftarrow a_i : a_i \in Dx_i$

3 Razonar hacia adelante (Forward checking):

4 Eliminar de los dominios de las variables (x_{i+1} , …, x_n) aún no instanciadas, aquellos valores inconsistentes con respecto a la instanciación (x_i , a_i), de acuerdo al conjunto de restricciones.

5 Si quedan valores posibles en los dominios de todas las variables por instanciar, entonces:

6 $i < n$, incrementar i e ir al paso 1

7 $i = n$, salir con la solución

8 Si existe una variable por instanciar, sin valores posibles en su dominio, entonces retractar los efectos generados por la asignación $x_i \leftarrow a_i$. Hacer:

9 Si quedan valores por intentar en Dxi, ir al paso 2

10 Si no quedan valores:

11 $i > 1$, decrementar i y volver al paso 2

12 $i = 1$, salir sin solución

Algoritmo 2: Pseudo código de Forward Checking

2.5 Estrategias de Enumeración.

Como ya se ha mencionado, lo más habitual para la resolución de un CSP es utilizar técnicas híbridas, la cual consiste en alternar fases de propagación y enumeración, en esta última se debe establecer el orden el cual se van a considerar las variables, así como el orden en el cual se van a instanciar los valores de los dominios de cada una de las variables [1]. Para establecer dichos órdenes se utilizan heurísticas de selección de variable y heurísticas de selección de valor respectivamente, las que en conjunto constituyen las denominadas Estrategias de Enumeración. En la literatura se ha establecido que seleccionar un orden correcto de las variables y de los valores puede mejorar considerablemente la eficiencia de resolución [1] [36] [37], es por esto que existen diversos esfuerzos en definir este tipo de heurísticas, algunas de las cuales se describen en la presente sección.

2.5.1 Heurísticas de Selección de Variable.

El orden en el cual las variables son asignadas durante la búsqueda puede tener un impacto significativo en el tamaño del espacio de búsqueda explorado. Esto puede ser contrastado tanto empírica como analíticamente.

Generalmente las heurísticas de selección de variables tratan de utilizar lo antes posible las variables que más restringen a las demás. La intuición es tratar de asignar lo antes posible las variables más restringidas y de esa manera identificar las situaciones sin salida lo antes posible y así reducir el número de vueltas atrás. La selección de variables puede ser estática o dinámica [1] [3] [38].

Selección Estática: Las heurísticas de ordenación de variables estática generan un orden fijo de las variables antes de iniciar la búsqueda, basado en información global derivada de la estructura de la red de restricciones original que representa el CSP. En un caso general, interesa encontrar el mejor orden para la asignación de variables. Entre las heurísticas más utilizadas podemos mencionar: Minimum Width, Maximum Degree, Mínimum Domain Variable.

Selección Dinámica: El problema de los algoritmos de ordenación de variables estáticos es que no tienen en cuenta los cambios en los dominios o relaciones de las variables causados por la propagación de las restricciones durante la búsqueda. Por ello, estas heurísticas generalmente se utilizan en algoritmos de comprobación hacia atrás (o look-backward) donde no se lleva a cabo la propagación de las instanciaciones que se van realizando.

Las heurísticas de ordenación de variables dinámica pueden cambiar el orden de selección de las variables de forma dinámica durante el proceso de búsqueda, cada vez que requiere la instanciación de una variable. La ordenación se basa en las instanciaciones ya realizadas y en el estado de la red en cada momento de la búsqueda.

Se han propuesto varias heurísticas de ordenación de variables dinámicas. Las más comunes se basan en el principio de primer fallo que sugiere que “para tener éxito deberíamos intentar primero donde sea más probable que falle”. De esta manera las situaciones sin salida pueden identificarse antes y además se ahorra espacio de búsqueda. De acuerdo a este principio, en cada paso, se selecciona la variable más restringida. Entre las más utilizadas podemos citar: Minimum Remaining Values, Maximun Cardinality.

2.5.2 Heurísticas de Selección de Valor.

En comparación con la selección de variables, se ha realizado menos trabajo sobre heurísticas para la selección de valores. Estas heurísticas tienen como objetivo seleccionar el valor más prometedor para cada variable en su dominio de instanciación. La idea básica es seleccionar el valor de la variable actual que más probabilidad tenga de llevarnos a una solución, es decir, identificar la rama del árbol de búsqueda que sea más probable que obtenga una solución. La mayoría de las heurísticas propuestas tratan de seleccionar el valor menos restringido de la variable actual, es decir, el valor que menos reduce el número de valores útiles para las futuras variables, o alternativamente, el que deja los dominios mayores. Esto sigue la intuición de que un subproblema es más probable que tenga solución cuantos más valores tengan las variables que quedan por instanciar en sus dominios. Entre las más utilizadas podemos mencionar:

- **min-conflicts**. Esta heurística ordena los valores de acuerdo a los conflictos que generan con las variables aún no instanciadas.
- **max-domain-size**. Alternativamente a la anterior, esta heurística selecciona el valor de la variable actual que deja el máximo dominio en las variables futuras.
- **weighted-max-domain-size**. Esta heurística especifica una manera de romper empates en el método anterior, en el caso de que existan varios valores de la variable actual que dejen el mismo máximo dominio en las variables futuras. Entonces, se basa en el número de futuras variables que tienen la mayor talla de dominio
- **point-domain-size.** Esta heurística asigna un peso (unidades) a cada valor de la variable actual dependiendo del número de variables futuras que se quedan con ciertas tallas de dominios.
- **Supervivencia**. Esta heurística propone una variación de la idea anterior de la heurística min-conflicts.
- **max-promise**. Se selecciona el valor con la máxima promesa. Al usar el producto en vez de la suma del número de valores compatibles, la heurística max-promise trata de seleccionar el valor que deja un mayor número de soluciones posibles después de que este valor se haya asignado a la variable actual.

CAPÍTULO 3

Lenguajes de Programación con Restricciones

3 Lenguajes de Programación con Restricciones.

En este capítulo se hace reseña a los dos paradigmas de programación más utilizados para resolver problemas mediante Programación con Restricciones. Tradicionalmente se han utilizado Lenguajes Lógicos para implementar el modelado y solución de CSP, dadas las características de eficiencia y declaratividad con la que estos lenguajes cuentan. Pero también existen diferentes esfuerzos por implementar resolver los CSP mediante otros lenguajes como es el caso de los Lenguajes Orientados a Objetos, para los cuales se han implementado librería especializadas para el manejo de CP. Estos esfuerzos por generar lenguajes de programación con restricciones suelen conocerse en el mundo de CP como Solvers, los cuales son compiladores o librerías especializadas para implementar CP.

3.1 Programación Lógica con Restricciones (CLP).

La programación lógica con restricciones (en ingles Constraint Logic Programming, CLP) [39] [9] [40] nació como una mezcla de dos paradigmas: la Programación con Restricciones (CP) [3] [41] y la Programación Lógica (Logic Programming, LP) [42], y su éxito radica en que combina la declaratividad de la Programación Lógica con la eficiencia de la Programación con Restricciones [1].

3.1.1 Programación Lógica.

En este informe no se pretende describir la programación lógica pues es un concepto ampliamente estudiado y conocido. En realidad se espera familiarizar, o al menos conocer conceptos básicos de LP. Aun así, se incluirá una breve descripción sobre este tipo de programación.

LP consta de un conjunto arbitrario de símbolos de predicado y de función, y el paradigma LP se basa en una idea declarativa donde los programas se construyen a partir de implicaciones lógicas entre colecciones de predicados [42]. Un programa lógico consiste en un conjunto de reglas, llamadas cláusulas, de la forma $H :- B$, donde H se denomina

la cabeza de la regla y $B = B_1, \ldots, B_n$ (n ¸ 0) el cuerpo de la regla (el cual se interpreta como una conjunción de átomos). Tanto H como B_i (para $1 \leq i \leq n$) son átomos con la forma $p(t_1, \ldots, t_m)$ (m¸ 0), siendo p un símbolo de predicado y los t_i's son términos, o sea, variables, constantes o una función de aridad m aplicada a m términos. Si n = 0, se dice que la cláusula H :-B es un hecho.

La idea intuitiva de una regla $H :- B$ es que, si la conjunción de $B_1, \ldots, B_n$ es verdadera, entonces H también es verdadero. H y B pueden contener variables y el significado de la cláusula es una implicación desde B a H interpretada de la siguiente manera: para todos los valores posibles que puedan tomar las variables que aparecen en H, existen valores que pueden tomar las variables que aparecen en B de tal forma que $B \rightarrow H$. Se observa que si la cláusula es un hecho, esto quiere decir que H es verdadero para cualquier valor de sus variables.

Operacionalmente hablando, ejecutar un programa lógico consiste en preguntar el valor de verdad (i.e., verdadero o falso) de cierta sentencia : $-G$, llamada el objetivo, que contiene una conjunción de átomos ($G = G_1, \ldots, G_n$). Esto se interpreta como "preguntar por los valores adecuados de las variables que aparecen en G para que la conjunción $G_1, \ldots, G_n$ sea verdad con respecto al programa lógico". La respuesta a un objetivo es un conjunto, posiblemente vacío si no hay solución, de substituciones (asignaciones de variables a términos) que hacen que, cuando cada variable es substituida por su valor correspondiente según la asignación, el objetivo G sea verdadero con respecto al programa.

Uno de los puntos clave en la resolución de objetivos es la existencia de un paso de unificación realizado en cada una de las etapas del proceso global. Este paso consiste en unificar, a través de una substitución σ, un átomo G_i, que es parte de un (sub-)objetivo :-G ($G = G_1, \ldots, G_n$) con la cabeza H de una cláusula, digamos, $H : -B$. En este paso, σ es el unificador mas general entre G_i y H, y el objetivo:-G, es reemplazado por un nuevo (sub-) objetivo (:− $(B, G_2, \ldots, G_n)\ \sigma$ para $i = 1$) en proceso de resolución.

Como ejemplo se considera el siguiente programa de una sola cláusula escrito en el lenguaje de programación lógica estándar, Prolog [43] [44]:

menor_que_tres(X): − integer(X), X < 3.

El objetivo :− menor que tres (2) devuelve la substitución {X=2} la cual hace que el cuerpo {integer(2), 2 < 3} sea verdad.

Muy básicamente, la resolución integra otras etapas, tales como el renombramiento de variables o la elección del átomo G_i en el objetivo, que no consideramos en esta sección. No se insistirá más en los detalles de notación de LP. Es fácil intuir que LP constituye un marco natural en el cual los CSPs pueden ser modelados a alto nivel pues las

restricciones son relaciones entre las variables y los programas lógicos contienen relaciones de variables en forma de cláusulas.

3.1.2 El esquema CLP.

A continuación presentamos el esquema básico de CLP. Este esquema fue presentado por Jaffar y Lassez en 1987 [8], y supone un marco formal, basados en restricciones, para las semánticas operacionales, lógicas y algebraicas de una clase extendida de programas lógicos. A pesar de que los programas de CLP dependen del dominio de aplicación (el dominio de cómputo), este esquema de CLP fue ideado para la creación de lenguajes lógicos compartiendo el mismo mecanismo de evaluación.

La idea básica en CLP es la de reemplazar la unificación clásica de LP por la resolución de restricciones (un resolutor) sobre un dominio de computo específico. Esta idea dio lugar al esquema CLP(X) descrito en [8]. En este esquema, X representa al dominio de cómputo sobre el cual las restricciones serán resueltas. Las diferentes instancias de X ($\mathcal{R}$, $\mathcal{Z}$ conjuntos, Booleanos, etc.) generan las diferentes instancias del esquema CLP(X) el cual permite además que CLP pueda resolver problemas que LP no puede por lo que es especialmente interesante para el paradigma declarativo.

Como ejemplo se considera el programa mostrado en el de la sección 3.1.1 Un objetivo tal como *:−menor_que_tres*(Y) devuelve, en general, un error de instanciación en cualquier lenguaje lógico puesto que la variable Y no esta ligada (a un termino, en este caso un valor entero) previamente a la realización del proceso de resolución. Sin embargo, si lanzamos este objetivo en un sistema lógico con restricciones de enteros, el objetivo devuelve un conjunto de múltiples respuestas indicado por la relación $Y \in [-\infty, 2]$.

En cada una de las instancias del esquema, el lenguaje de programación lógico subyacente es extendido con un conjunto de operaciones y estructuras que pueden ser usadas sobre el dominio de computo y que pueden ser directamente utilizadas por el usuario.

El esquema CLP(X) se caracteriza asimismo por un mecanismo de resolución de las restricciones inmerso en el lenguaje lógico subyacente. Este mecanismo, llamado el resolutor de restricciones, constituye un procedimiento de decisión capaz de comprobar si una restricción o conjunto de restricciones puede ser satisfecha. En el proceso clásico de resolución de objetivos en LP, el resolutor reemplaza a la unificación estándar por un algoritmo que decide la satisfiabilidad de las restricciones, donde la satisfiabilidad de las restricciones significa decidir la consistencia de las mismas. La mayoría de los resolutores actuales inmersos en lenguajes lógicos son incompletos por lo que suelen estar definidos mediante algún algoritmo de propagación de restricciones [45].

En un contexto general, podemos decir que un lenguaje de CLP puede considerarse un lenguaje lógico donde se han incorporado restricciones y métodos de resolución de restricciones. En este sentido, la diferencia entre LP y CLP estriba en la interpretación operacional de las restricciones y no en su interpretación declarativa.

Resumiendo, la idea crítica del esquema CLP es que, tanto el lenguaje lógico subyacente como sus semánticas operacionales y declarativas pueden ser parametrizadas por un dominio de computo X y las operaciones sobre este dominio.

3.1.3 Aplicaciones de CLP.

CLP ha sido aplicada con bastante éxito en la resolución de problemas reales como por ejemplo aplicaciones clásicas resueltas por técnicas de Investigación Operativa como la búsqueda combinatorial [48], problemas de "cutting stock" [47] y problemas de planificación [49]. Otras áreas donde CLP ha demostrado su potencia son la diagnosis de fallos en circuitos electrónicos, redes neuronales, aplicaciones industriales, visualización de relaciones con datos biológicos, manejo de robots, reconstrucción de secuencias originales de ADN a partir de fragmentos de particiones de enzima y reconstrucción tridimensional de modelos de proteínas entre otras muchas aplicaciones. Más información sobre las aplicaciones practicas de CLP puede encontrarse en [50].

3.1.4 Lenguajes de Programación con Restricciones Lógicos.

En esta sección, se presenta una pequeña descripción de diferentes lenguajes de programación lógicos Estos lenguajes se clasifican en dos grupos; los de Glass Box (Caja Transparente) y los Black Box (Caja Negra). Así pues, primero se explicará las características para cada clasificación.

3.1.4.1 Glass Box vs Black Box.

La distinción entre un lenguaje Glass Box y Black Box no está siempre clara. Los lenguajes de Glass Box[51] proporcionan restricciones muy simples y primitivas, cuyo esquema de propagación puede ser formalmente especificado. Estas restricciones pueden ser usadas para construir restricciones de alto nivel especializadas, adecuadas para cada aplicación. Alternativamente, los lenguajes Black Box son los que proporcionan una amplia gama de restricciones de alto nivel cuya implementación se oculta al usuario. Estas restricciones realizan tareas específicas muy eficientemente. En estos lenguajes, es difícil que un usuario agregue nuevas restricciones puesto que tales restricciones tienen que ser definidas a bajo nivel requiriendo un conocimiento detallado de la implementación.

3.1.4.2 Lenguajes Glass Box.

Hay dos tipos diferentes de lenguajes Glass Box. La diferencia radica en la forma en que la propagación de restricciones pueden ser definidas: Una forma es usando construcción relacional llamado Índices (en ingles *Indexical*) [8] o por manejo de reglas de restricciones (en ingles *Constraint Handling Rules,* CHRs)[53].

Lenguajes con Índices.

Algunos lenguajes que soportan índices son clp(FD), Sictus y IF/Prolog. Un índice es una regla reactiva funcional de la forma X en R donde X es una variable de dominio. R es una expresión de rango de la forma $f_1, \ldots, f_2$ cuyos términos f_1 y f_2 son rangos singulares, parámetros, enteros, combinación de términos usando operadores aritméticos en rangos por índices. Las formas permitidas para un rango por índices depende del lenguaje pero, normalmente, es una de las siguientes:

- min(Y): representa el mínimo valor de la variable de dominio Y.
- max(Y): representa el máximo valor de la variable de dominio Y.
- val(Y): representa en valor de Y tan pronto como es establecido.
- dom(Y): representa el dominio actual de Y.

Las restricciones con índices pueden ser vistas como una maquina abstracta para la resolución de restricciones basada en propagación [14]. Es posible una codificación directamente a muchas restricciones de alto nivel (en FD) con estas restricciones básicas.

Una pequeña descripción de estos lenguajes se da a continuación:

SICStus: Permite dos modos: el primero, "iso mode" cumple estrictamente la norma ISO/IEC 13211-1 (1995) que estandariza el lenguaje Prolog a nivel internacional. En el segundo modo de trabajo, "sicstus mode", se añaden extensiones que dotan a este intérprete de mayor potencia. Tiene un entorno de desarrollo con facilidades para hacer interfaces gráficas.

clp(FD): Básicamente trata solo con restricciones del tipo X en R (R no solo tiene que ser de tipo {1..10} sino que puede ser indexado. Pueden crearse restricciones de alto nivel (restricciones de usuario). Cada restricción especifica como una variable restringida es actualizada cuando el dominio de otra variable cambia. Es una extensión de Prolog que apoya la forma de especificar las variables de dominio, las limitaciones, y estrategias para instanciar variables. En general, un CLP (FD) del programa está constituida por tres partes: la primera parte, llamada variable de generación, genera variables y especifica sus dominios, la segunda parte, llamada la generación de limitación,

especifica las restricciones en las variables, y la última parte, llamada etiquetado, instancia las variables para hacer la enumeración.

IF/Prolog: Otro sistema que cumple la ISO para Prolog y que tiene interfaces con Java, C/C++, y bases de datos relacionales. Pero es un lenguaje puramente lógico.

Lenguajes con reglas de manejo de restricciones (CHR).

Para CHR (en ingles constraint handling rules) se usa una librería que está construida sobre Eclipse, añadiendo las reglas para manejo de restricciones. Una regla de manejo de restricciones puede definir unas simplificaciones y propagaciones sobre restricciones definidas por el usuario.

Una regla de simplificación sustituye restricciones por otras más simples mientras conserven la equivalencia lógica. Por ejemplo:

X > Y; Y > X <==> false

Una regla de propagación añade nuevas restricciones que son lógicamente redundantes, pero puede provocar futuras simplificaciones. Por ejemplo:

X > Y; Y > Z ==> X > Z

Aplicar repetidamente reglas de manejo de restricciones incrementará la simplificación y, posiblemente, resolverá las restricciones definidas por el usuario. Obsérvese que CHRs fue pensado originalmente como lenguaje para la simplificación de restricciones pero ha demostrado desde entonces ser útil para construir soluciones a restricciones para aplicaciones y dominios particulares.

3.1.4.3 Lenguajes Black Box.

OZ: Basado en la programación funcional de orden superior y la programación lógica con restricciones. Combina funciones con relaciones.

Oz proporciona algoritmos para decidir la satisfacibilidad e implementación de restricciones básicas con la siguiente forma: X = n; X = Y o X : : D donde X e Y son variables, n es un entero no negativo y D es un dominio finito. Las restricciones básicas residen en el almacén de restricciones. Las restricciones no básicas, como X + Y = Z, no están contenidas en el almacén pero son impuestas por propagadores [56]. Un propagador Oz es un agente que lee en el almacén de restricciones e intenta reducir los dominios fijados ahí añadiendo restricciones básicas al almacén.

Por ejemplo, suponiendo que hay un almacén de restricciones que contiene las variables X, Y con dominio {1,…, 10}. El propagador para X + Y = 5 reduce el dominio de X e Y a {1,..., 4}. El propagador X + Y = 5 restringe las variables X e Y. Añadiendo la restricción Y = 1 reduce el dominio de Y a 1 y el dominio de X a 4.

Los propagadores son provistos por ambos lenguajes Glass y Black Box. Sin embargo en los lenguajes Glass Box, las indexaciones y CHRs son componentes básicos y los propagadores se construyen a partir de ellos. Por otro lado, en los lenguajes Black Box no hay medios de especificar directamente la propagación de las restricciones de modo que los propagadores tales como + o – son primitivos por si mismos y son determinados solamente por su semántica operacional.

Oz tiene una semántica formal simple y una implementación eficiente, el Sistema de Programación Mozart. Oz es un lenguaje orientado a la concurrencia, término introducido por Joe Armstrong, el principal diseñador del lenguaje Erlang. Un lenguaje orientado a la concurrencia hace a la concurrencia fácil de usar y eficiente. Pertenece a la familia de los CCPLs (Concurrent Constraint Programming Languages, CCPLs).

Eclipse: Eclipse es un entorno de programación que contiene el lenguaje de programación PROLOG y algunas extensiones que permitirán manejar bases de datos (incluyendo bases de datos declarativas y bases de conocimiento), programación lógica basada en restricciones (en concreto programación lógica basada en restricciones sobre dominios finitos), y programación concurrente. Eclipse incluye las restricciones en dominio finito tradicionales. Permite además la escritura de extensiones como restricciones definidas por el usuario o resolutores completamente nuevos como CHR.

Estas extensiones están basadas en un mecanismo de suspensión y reanimación de objetivos proporcionados por Eclipse. Para hacer una extensión, el usuario necesita un buen conocimiento del funcionamiento interno del sistema y es por esta razón que Eclipse no es catalogado como lenguaje Glass Box. Note se que además de enteros, la librería de dominios finitos (FD) permite elementos atómicos (atoms, strings, floats) y elementos compuestos básicos (f(a,b)).

En 1984 se creo el ECRC (European Computer-Industry Research Centre, ECRC) por tres compañías Europeas para explorar el desarrollo de técnicas avanzadas de resolución aplicables a problemas prácticos. En particular tres sistemas de programación fueron designados e implementados. Primero un problema complejo puede ser resuelto con un hardware multiprocesador y eventualmente en una red de maquinas. Segundo soporte de avanzadas técnicas de base de datos, para un procesamiento inteligente en aplicaciones extensas en datos. Tercero se estableció el uso de CHIP un lenguaje de programación lógica.

En 1991 Los tres sistemas se mezclaron generando Eclipse. El cual estaba inicialmente basado en CHIP. Durante los 15 años próximos de solvers de programación con restricciones y los interfaces de los solvers soportados por Eclipse fueron continuamente extendidos para responder a los requerimientos de los usuarios.

El primer interfaz liberado con un estado del arte lineal y mezclado con programación entera fue entregado en 1997. La integración de solver sobre dominios finitos y solvers de programación lineal, soportando algoritmos híbridos se realizó en el 2000. En 2001 Se creo la librería *ic*. Que soporta restricciones booleanas, enteras y reales que cubre las demandas reales del uso practico: escalable, robusto y ortogonal. Eclipse también incluye las librerías de algunos otros lenguajes de programación lógica, por ejemplo CLP(R), que fue desarrollado de forma separada. Por otro lado con las facilidades de resolución de restricciones, la programación paralela y las facilidades de bases de datos de Eclipse han sido mucho mas usadas, por otro lado por años algunas funcionalidades han sido cortadas del sistema. En Agosto de 2004 Eclipse fue adquirida por Cisco System.

Eclipse es usado por múltiples instituciones para educación investigación por todo el mundo. Siendo explotado para múltiples propósitos; planeación de producción, scheduling, Biomática, etc. Incluso en software de Cisco [71].

3.2 Programación Orientada a Objeto con Restricciones.

La Programación con Restricciones involucra tanto el modelamiento como la resolución de sistemas basados en restricciones. Estos problemas en su mayoría han sido implementados utilizando Lenguajes de Programación Lógicos. Esto no implica que sea el único paradigma bajo el cual se pueda implementar la programación con restricciones, también se puede utilizar Lenguajes Estructurados y Orientados a Objetos. Un enfoque declarativo multiparadigma con restricciones es el ofrecido en el lenguaje LIFE [58] que es un lenguaje experimental que propone integrar la Programación Lógica, la Programación Funcional, la Programación Orientada a Objetos y un mecanismo de resolución sobre un dominio ordenado de árboles de características (features), admitiéndose restricciones tales como la igualdad (p.e. la unificación) y el emparejamiento (p.e, matching) sobre términos de características (features). El cual es el precursor de otros lenguajes tales como LOGIN [59] y Le Fun [60].

Actualmente lo que se tiene es integrar librerías especificas basadas en Lenguajes Orientados a Objetos, principalmente C++, estas librerías cuentan con características especiales, para definir restricciones y realizar backtracking

3.2.1 Programación Orientada a Objetos.

Según Booch: "*La programación Orientada a Objetos es un método de implementación en el cual los programas están organizados como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas vía relaciones de herencia*".

Cuando se habla de programación a objeto, se consideran tres elementos importantes; usa objetos, no algoritmos, como bloques lógicos de construcción (jerarquía "parte de…"). Cada objeto es instancia de alguna clase, las clases están relacionadas entre si vía relaciones de herencia (jerarquía "tipo de…").

Un programa puede parecer orientado a objetos, pero si alguno de los tres elementos falta, no lo es. Específicamente, la programación sin herencia no es orientada a objetos, más bien es programación con tipos abstractos de datos.

Hay principalmente cinco estilos (paradigmas) de programación: Modelo Tipo de Abstracción, Orientado a procedimientos Algorítmico, Orientado a objetos Clases y objetos, Orientado a metas lógicas, Expresadas en cálculo de predicados, Orientado a reglas reglas if-then, Orientado a restricciones relaciones invariantes

No hay un único estilo de programación adecuado para todo tipo de aplicaciones. Cada uno de estos estilos tiene su propio marco conceptual, su manera diferente de pensar acerca del problema. Para el Paradigma Orientado a Objetos (OO), el marco conceptual es el modelo de objetos. Hay cuatro elementos principales en este modelo:

Abstracción: Las abstracciones son una manera fundamental que tenemos los humanos de manejarnos con la complejidad. Estas devienen del reconocimiento de las similaridades entre ciertos objetos, situaciones o procesos en el mundo real, y la decisión de concentrarse en estas similaridades ignorando las diferencias. Una abstracción es una descripción simplificada de un sistema que enfatiza algunos de sus detalles o propiedades mientras suprime otros. Una buena abstracción es la que enfatiza detalles que son significantes al lector, y suprime los que no lo son.

Encapsulación: El encapsulamiento (ocultamiento de la información) previene que los clientes vean la parte interna, donde el comportamiento de la abstracción es implementada. Entonces, según Booch: "El encapsulamiento es el proceso de ocultar todos los detalles de un objeto que no contribuyen a sus características esenciales.

Modularidad: El módulo es una construcción separada. En estos lenguajes, las clases y los objetos forman la estructura lógica de un sistema; colocando estas abstracciones en módulos se produce la estructura física. La mayoría de los lenguajes que soportan al módulo como un concepto separado, también distinguen entre la interfase y la implementación del módulo.

Jerarquía: Un conjunto de abstracciones frecuentemente forman una jerarquía, e individualizándolas en nuestro diseño, simplificaremos el entendimiento del problema. Con esto, La jerarquía es un ordenamiento de abstracciones.

Las dos jerarquías más importantes en un sistema complejo son su estructura de clases (jerarquía tipo de) y la estructura de objetos (jerarquía parte de).

3.2.2 Lenguajes de programación con restricciones Orientados a Objetos.

A continuación se presentan algunos de los lenguajes que hemos clasificado como Orientado a Objetos, en su mayoría son librerías implementadas en C++ y Java.

Ilog SOLVER:

Ilog SOLVER es una librería de C++ para programación con restricciones; por lo que los datos y las estructuras de control deben ser definidos en C++. Este no es estrictamente un sistema CLP, sin embargo tiene usos aproximados de CLP y es un sistema comercial popular para la resolución de problemas de satisfacción de restricciones.

En Ilog SOLVER, una restricción puede ser un objeto o una expresión booleana con valores falsos (IlcFalse) o true (IlcTrue). El valor depende de la capacidad de satisfacer la restricción. Si la restricción no puede ser violada, entonces a expresión esta limitada a IlTrue y si la expresión no puede ser satisfecha entonces esta limitada a IlcFalse. Estas expresiones pueden combinarse con operadores lógicos (and, or y not) para crear restricciones más complejas.

Cuando una restricción es postergada (usando la función IlcPost), la restricción es usada inmediatamente para reducir los dominios de las variables restringidas que involucra. El Backtracking es provisto combinando elementos no deterministas. Ilog Solver está desarrollado pensando en la eficiencia.

B-Prolog:

Si bien este es un lenguaje lógico, lo hemos encasillado en Orientado Objeto, por su manejo de interfaz para C y Java. Como con Eclipse, su sistema proporciona un conjunto de predicados de dominio finito como los aritméticos o las restricciones booleanas y un conjunto de primitivas para procesar las variables de dominio. Este conjunto de predicados de restricciones contenidos es menor para el B-Prolog que el que a su vez provee Eclipse. Además provee un interfaz bidireccional con C y Java.

GeCode:

GeCode (en ingles Generic Constraint Development Enviroment) es una biblioteca en C++ este lenguaje de programación propuesto inicialmente en el año 2002, pero su primera versión se libero el 2005. Su Autor Christian Schulte se desempeña como profesor asociado en el departamento de electrónico, informática y sistemas en el instituto real de tecnología de Estocolmo Suecia.

GeCode se caracteriza por ser de código abierto, una arquitectura abierta, libre, portátil, que provee un medio ambiente para el desarrollo de sistemas y aplicaciones basados en restricciones. GeCode se aplica en C++ que se rige al lenguaje C++ estándar. Puede ser compilado con modernos compiladores de C++. Actualmente es distribuida bajo licencia tipo BSD [61] [64]. GeCode dispone del uso de propagación de restricciones y búsquedas completas para cubrir los espacios de solución, los cuales trabajan sobre dominios finitos. Otro elemento destacable es que con los propagadores se implementan las restricciones.

Desde la primera versión liberada en el 2002 se han liberado otras 12 versiones, la versión 1.0.0 el 6 de diciembre del 2005, la versión 2.0.0 en noviembre del 2007 y la versión 2.1.1 en marzo del 2008 y próximamente la liberación de la versión 3.0.0.

En cuando a la documentación, solo se poseen los documentos de usuario de sistema, y una serie de publicaciones, de los autores y algunos experimentos y comparaciones realizados. En cuanto al uso de GeCode, su uso principalmente se ha dado en universidades, para proyectos de educación e investigación, pero varias empresas han incorporado el uso de GeCode. Posee distintos niveles de abstracción que se observa en la figura 3.1.

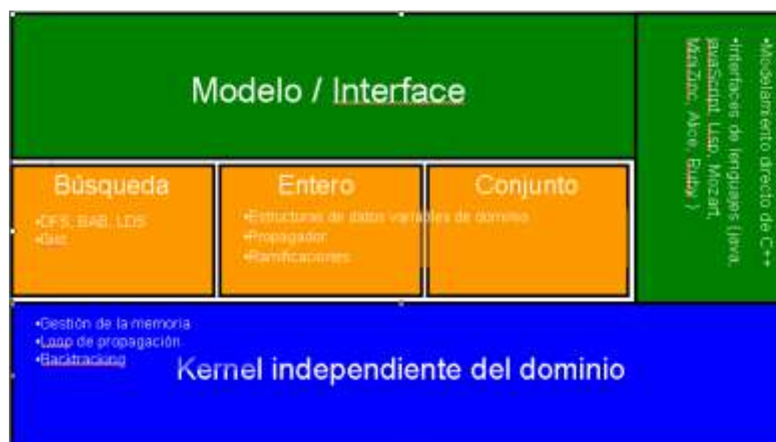

Figura 3.1: Niveles de abstracción de FECODE

GeCode permite la utilización de código genérico utilizando plantillas de C++, permitiendo optimizaciones del compilador, también se puede generar su propio código cuando las plantillas no son suficientes. En cuanto a la ejecución puede ser difícil dar detalles de ella, existiendo un trade-off entre eficiencia y privacidad.

GeCode/J:

Es un interfaz Java para GeCode, el cual permite extrapolar todas las potenciabilidades de GeCode en C++, con las características portables de Java. Además incluye la representación de árboles de búsqueda gráficos como lo hace Oz [62][65]. Es un mapeo de GeCode, el cual no es una simple capa de modelo como Gecode/R, es un esfuerzo por mantener la misma representatividad y los conceptos entre GeCode y GeCode/J. Una de las ventajas de este estrecho mapeo es que trata de escribir de manera simple propagadores y ramificaciones directamente en Java.

Una de las principales razones, por las cuales se implemento GeCode en java fue principalmente por motivos educativos, en los cuales se necesitaba algún tipo de interfaz grafica.

Choco:

Choco es una biblioteca Java para Problemas de Satisfacción de Restricciones (CSP), programación con restricciones (CP). Está construido sobre mecanismo de propagación basado en eventos con estructuras de backtracking [63]. Se ha utilizado principalmente para: la enseñanza (un solucionador de restricciones orientado al usuario con código fuente abierto), La investigación (para el estado del arte de algoritmos, técnicas y restricciones definidas por el usuario, dominios y variables), Las aplicaciones de la vida real (aplicación integradas con choco).

CAPÍTULO 4

Estudio Comparativo

4 Estudio comparativo.

Es evidente el éxito de paradigma de programación lógica con restricciones (CLP) [7] [8] [9], ya que encontramos que se ha incrementado el numero de lenguajes CLP que ahora son usados para muchas aplicaciones de la vida real [51]. Hay dos razones principales para este éxito: primero, los CLP extienden el paradigma de programación lógica volviéndola mas declarativa y las soluciones mas legibles, y en segundo lugar, apoya la propagación de las restricciones para los dominios específicos, proporcionando una implementación eficiente para procedimientos de cómputo costosos. Pero estas características también pueden ser adoptadas por otros lenguajes, para este caso específico con lenguajes orientados a objeto, los cuales también pueden presentar una forma clara de implementar las restricciones y podrían definir mejor los propagadores, aumentando la eficiencia. Además pueden incorporar características gráficas que permitan visualizar mejor la generación de soluciones [65].

Sin embargo, los sistemas se diferencian perceptiblemente en cómo las soluciones se pueden expresar y la eficacia de su ejecución. Es importante que estos dos factores sean considerados al elegir el mejor lenguaje para un uso particular. De hecho, la selección incorrecta de un lenguaje puede ser desastrosa, no solamente por la eficiencia en su funcionamiento, sino también con respecto a la claridad del código de la solución, importante para las modificaciones futuras.

De los dominios, el dominio finito (FD) [6] es uno de los más estudiados, el cual proporciona un marco de trabajo adecuado para solucionar problemas discretos de satisfacción de restricciones. El FD es particularmente útil para modelar problemas tales como scheduling, planeación, embalaje, timetabling, por consiguiente la mayoría de los sistemas proporcionan bibliotecas substanciales de FD. En este trabajo solamente se consideran dominios finitos (FD) y ó casos de dominio booleano, los que pueden verse como un caso particular de dominio finito.

El objetivo principal de este apartado es poder generar un estudio que permita establecer un medio parametrizable para poder comparar los distintos solvers que se puedan utilizar. Este medio es una plantilla de comparación, para la cual se definen una serie de criterios. La selección de estos criterios se justifican en esta sección usando como base las características deseables para cada solver dentro el proceso de resolución de un CSP.

La resolución de un CSP inicialmente necesita de una fase de modelización en la cual es importante definir variables, dominios, restricciones, en definitiva lo que llamaremos expresividad. Característica deseada para los lenguajes.

Por otro lado se necesita resolver el CSP en cuyo caso lo que se busca conocer y evaluar es la técnica de resolución utilizada por el lenguaje, lo cual va a influir directamente en la eficiencia del lenguaje para resolver CSPs.

La resolución del CSP esta dada por fases de propagación y enumeración, en la primera nos centraremos en la facilidad para poder trabajar en la reducción de los dominios de cada variable y en el caso de la fase de enumeración nos tenemos que centrar específicamente en las funciones de selección de variables y en las funciones de selección de valor para dichas variables. Estas etapas se pueden observar en la figura 4.1.

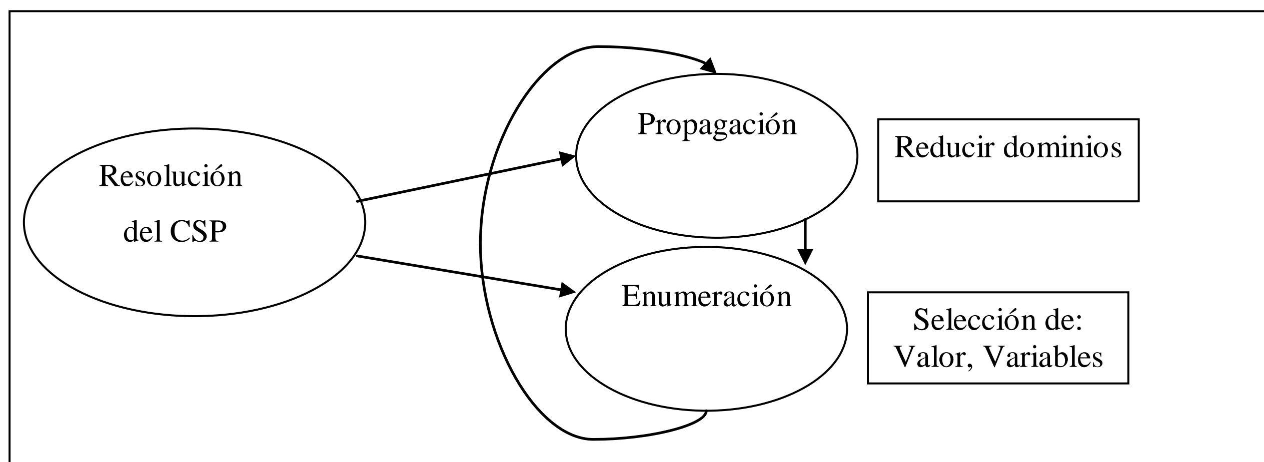

Figura 4.1: Etapas de resolución de CSP

Estas características deseables en los solver son consideradas para realizar la plantilla de comparación y principalmente para seleccionar los criterios o indicadores de comparación. Ahora bien no solo se van a considerar características internas de cada solver, sino también características externas que ayuden a generar una visión global para comparar los distintos lenguajes de programación con restricciones las cuales serán definidas en las siguientes secciones.

4.1 Definición de criterios.

Para poder realizar una comparación entre lenguajes se deben establecer una serie de criterios, puntos que nos permita realizar una correcta comparación de lenguajes. Para establecer los criterios inicialmente se tiene que hacer una clasificación de estos, ya que como se dijo en la sección previa no solo se compararán criterios internos, sino que también criterios externos. Además se debe especificar que criterios están sujetos a prueba experimental y cuales a justificación bibliográfica.

4.1.1 Criterios Internos.

Los criterios internos buscar representar aquellas características o atributos del lenguaje propios a su estructura interna y de cómo son capaces de representar y permitir resolver los problemas de CSP. No se debe perder de vista que todo lenguaje de programación con restricciones debe ser simple, expresivo y eficiente además de entregar abstracciones que faciliten la reutilización y el desarrollo de restricciones para la implementaciones.

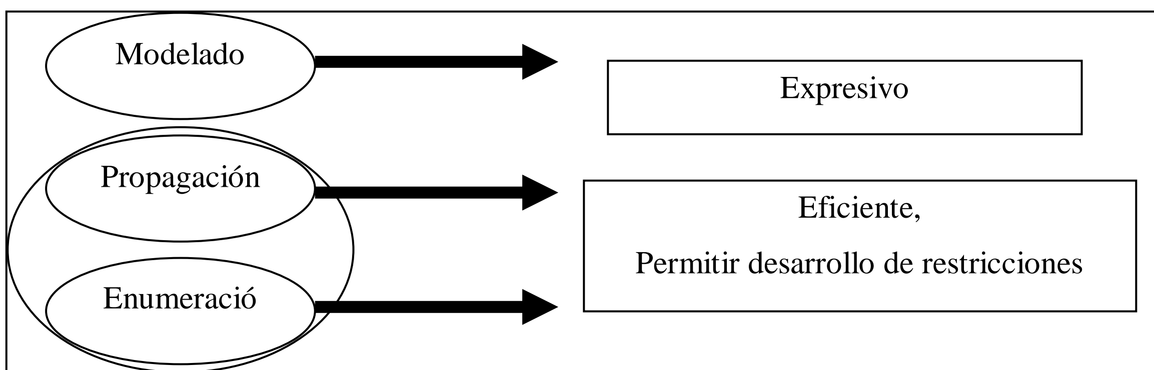

Figura 4.2: Características deseables en las etapas de modelado y resolución

De acuerdo a lo indicado en la figura 4.2 podemos generar dentro de los atributos internos una subclasificación Según expresividad, eficiencia y desarrollo de restricciones. Estos criterios o puntos de comparación se presentan a continuación y además se indica cuales están sujetos a prueba experimental, ósea cuales deben ser probados resolviendo un problema particular, y cuales están sujetos a justificación bibliográfica, ósea se pueden comprobar solo por medio de la investigación de medios bibliográficos; manuales, textos, publicaciones, etc.

4.1.1.1 Expresividad.

Se puede traducir el modelo fácilmente al lenguaje y se pueden comprender las restricciones del modelo directamente en el lenguaje implementado. Es difícil medir la expresividad de un lenguaje, pero, para hacerlo, los programadores simplemente escogen un problema específico y lo resuelven usando diferentes lenguajes. El lenguaje más expresivo será aquel en el que la solución al problema se puede expresar con mayor claridad, menor complejidad o menos palabras. Por eso, todos los lenguajes de programación con restricciones siempre están orientados a resolver problemas específicos. A continuación se presentan algunos criterios que pueden ser valiosos para medir la expresividad en un Lenguaje de Programación con Restricciones. Aclarando en que casos se usará una justificación experimental y en cuales solo se basara en una referencia bibliográfica.

Justificación Experimental:

Criterio: Líneas de código LOC: LOC; las líneas de código es una medida básica para la mayor parte del software, ambiguo. Su significado varía de un lenguaje de programación a otro, pero también dentro de un mismo lenguaje de programación. Por eso la cantidad de líneas de código se deben medir en la implementación de un determinado problema.

Justificación Bibliográfica:

Criterio: Tipificación: Un sistema de tipificación define cómo un lenguaje de programación clasifica los valores y las expresiones en tipos, cómo se pueden manipular estos tipos y cómo interactúan. Un tipo indica un conjunto de valores que tienen el mismo significado genérico o propósito (aunque algunos tipos, como los tipos de datos abstractos y tipos de datos función, tal vez no representen valores en el programa que se está ejecutando). Los sistemas de tipificación varían significativamente entre lenguajes, siendo quizás las más importantes variaciones las que estén en sus implementaciones de la sintáctica en tiempo de compilación y la operativa en tiempo de ejecución.

El proceso de verificar e imponer los límites impuestos por los tipos de datos –chequeo de tipificación- puede ocurrir tanto en tiempo de compilación (un chequeo estático) o en tiempo de ejecución (un chequeo dinámico). Si un lenguaje impone fuertemente las reglas de tipificación (es decir, generalmente permitiendo solo las conversiones de tipo de dato automáticas que no hagan perder información), uno se puede referir al proceso como fuertemente tipificado; sino, débilmente tipificado.

Criterio: Tipos de datos y estructuras: Un tipo de dato es un atributo de una parte de los datos que indica al computador (y/o al programador) algo sobre la clase de datos sobre los que se va a procesar. Esto incluye imponer restricciones en los datos, como qué valores pueden tomar y qué operaciones se pueden realizar. Tipos de datos comunes son: enteros, números de coma flotante (decimales), cadenas alfanuméricas. Las estructuras de datos son una colección de datos que se caracterizan por su organización y las operaciones que se definen en ella. Los datos de tipo estándar pueden ser organizados en diferentes estructuras de datos: estáticas y dinámicas.

Criterio: Definición de dominios: El dominio es el conjunto de valores posibles para una variables, en este sentido lo que se desea es buscar y comparar aquellas funciones o abstracciones que permitan definir los dominios posibles para una o mas variables para resolver un problema de CSP. Ya hemos dicho que nos vamos a evocar principalmente a los dominios finitos, en sus distintos tipos.

Criterio: Definición de restricciones: Para entender la definición de una restricción daremos a conocer algunas de las propiedades básicas.

La aridad de una restricción es el numero de variables que componen dicha restricción. Una restricción unaria es una restricción que consta de una sola variable. Una restricción binaria es una restricción que consta de dos variables. Una restricción ternaria consta de tres variables. Una restricción no binaria (o n-aria) es una restricción que involucra a un número arbitrario de variables.

Por ejemplo. La restricción $x \leq 5$ es una restricción unaria sobre la variable x. La restricción $x4 - x3 \neq 3$ es una restricción binaria. La restricción $2x1 - x2 + 4x3 \leq 4$ es una restricción ternaria. Por ultimo un ejemplo de restricciones n-aria seria $x1 + 2x2 - x3 + 5x4 \leq 9$.

Dado las características de las restricciones también es un buen punto de comparación entre los diferentes lenguajes de programación con restricciones, evaluar la capacidad de los lenguajes para poder definir dichas restricciones, de manera simple y expresiva, ojala expresadas de la manera más natural posible. Por ejemplo el uso de all_different ().

Criterio: Operaciones sobre dominios: Se buscará comparar los lenguajes en términos del uso de operadores, para trabajar sobre los dominios de una variable, en términos de los mecanismos que provea el lenguaje para operar sobre los dominios. Una operación de dominio apoya simultáneos accesos o actualizaciones a múltiples valores de un dominio de una variable. En muchos lenguajes esto es proporcionado por el apoyo a de un conjunto de tipos de datos abstractos para los dominios de las variables, como por ejemplo en Choco [66], Eclipse [67], Mozart [56], y Sicstus [68]. ILOG Solver, sólo permite el acceso por iteraciones sobre los valores del dominio de una variable.

Criterio: Integración: Se entiende por integración a la capacidad que posee un lenguaje de programación para poder conectar o integrar el solver con otros recursos de programación o módulos externos que pueden ser ajenos al lenguaje con el cual se está trabajando como por ejemplo: interfaces gráficas, otros leguajes o librerías compartidas. Una característica de esto es lo que puede ocurrir en las tecnologías .NET en las cuales existe una gran capacidad de integración e interconexión entre distintos lenguajes que cumplen con este Framework.

Criterio: Sencillez: Respecto a las cualidades deseables de todo lenguaje, la sencillez hace referencia a que, a veces, la forma encontrar la solución un problema puede llevar a escribir un algoritmo muy complejo, afectando a la claridad del mismo. Por tanto, hay que intentar que la solución sea sencilla, ósea permita implementar algoritmos claros y comprensibles usando un lenguaje simple. Algunos elementos que nos permiten comparar la sencillez de un lenguaje pueden ser: manejo de punteros y de referencias, uso de registros de tipo (*Struct*), el uso de definición de tipos (Typedef) y la de macros (#define), operaciones de manejo de memoria como liberar o eliminar memoria en tiempo real. [73].

4.1.1.2 Eficiencia.

Al hablar de eficiencia nos referimos a que el Lenguaje de Programación con Restricciones debe consumir la menor cantidad de recursos posible. Normalmente al hablar de eficiencia se suele hacer referencia al consumo de tiempo y/o memoria. Realiza un buen uso de los recursos programación y de sistema. A continuación se presentan algunos criterios que pueden ser valiosos para medir la eficiencia en un Lenguaje de Programación con Restricciones. Aclarando en que casos se usará una justificación experimental y en cuales solo se basara en una referencia bibliográfica.

Justificación Experimental:

Criterio: Tiempo de ejecución: Período en el que un programa es ejecutado por el sistema operativo. El período comienza cuando el programa es llevado a la memoria primaria y comienzan a ejecutarse sus instrucciones. El período finaliza cuando el programa envía la señal de término (normal o anormal) al sistema operativo. Este tiempo normalmente lo mediremos hasta obtener la primera solución en la resolución de un CSP y lo expresaremos en segundos.

Criterio: Uso de memoria: La memoria RAM es la memoria desde donde el procesador recibe las instrucciones y guarda los resultados. Es el área de trabajo para la mayor parte del software de un computador. Se debe medir la cantidad de memoria RAM utilizada de manera practica por una determinada instancia corrida bajo un solver en estudio. La mediremos en kilobytes.

Criterio: Uso de Disco: Al escribir un determinado programa en un lenguaje de programación este tendrá que estar alojado en una unidad de almacenamiento secundario, el cual puede ser un disco duro. Dependiendo del lenguaje de programación que se utilice se generarán más o menos archivos para representar dicho programa y estos archivos pueden utilizar más o menos cantidad de disco. Este criterio debe ser medido y comparado para el mismo problema en dos o más lenguajes diferentes midiendo la cantidad de archivos que se generan y el uso de memoria de almacenamiento, el cual puede estar medido en alguna unidad de almacenamiento como por ejemplo kilos o mega bytes.

Criterio: Nº backtrack: Backtrack es un algoritmo general para encontrar todas (o algunas soluciones) a un problema computacional, que construye incrementalmente los candidatos a las soluciones, y abandona cada candidato parcial *c* tan pronto como se determina que c no es posible que se convierta en una solución válida. El número de backtrack corresponde al número de veces que se tuvo que volver atrás ósea cambiar por otro candidato *c* para buscar una solución. Este criterio debe ser medido a través de la implementación de una instancia de prueba.

Justificación Bibliográfica:

Criterio: Reutilización: La reutilización de código se refiere al comportamiento y a las técnicas que garantizan que una parte o la totalidad de un programa existente se pueda emplear en la construcción de otro programa. De esta forma se aprovecha el trabajo anterior, se economiza tiempo, y se reduce la redundancia. Por lo general se deja el código reusable en un único lugar, llamándolo desde los diferentes programas. Este proceso se conoce como abstracción. La abstracción puede verse claramente en las bibliotecas compartidas, en las que se agrupan varias operaciones comunes a cierto dominio para facilitar el desarrollo de programas nuevos. Hay bibliotecas para convertir información entre diferentes formatos conocidos, acceder a dispositivos de almacenamiento externos, proporcionar una interfaz con otros programas, manipular información de manera conocida.

Criterio: Seguridad de tipos: La seguridad de tipos es el grado en que un lenguaje de programación desalienta o impide los *errores de tipo*. Un *error de tipo* es un error causado por una discrepancia entre diferentes tipos de datos. La seguridad de tipos es considerada una característica de los lenguajes. Es decir algunos lenguajes tienen incorporado algunos manejos de seguridad de tipos, para detectar estos errores si es que los programadores han implementado con estos errores. La seguridad de tipo podría afectar la eficiencia del lenguaje en procesos de compilación, motivo por el cual ha sido clasificado en este ítem.

Criterio: Tipo de traducción: El proceso de traducción consiste en tomar el código fuente y luego transformarlo a un lenguaje legible para la maquina ósea este debe ser traducido a binario para que las instrucciones que contienen puedan ser entendidas y ejecutadas por la máquina. Existen dos formas de llevar el código fuente a un código ejecutable; la compilación y la interpretación. La idea es comparar los lenguajes en función del uso de compilador o traductor para trasformar el código fuente. Este criterio también afecta enormemente la eficiencia que podría tener un lenguaje de programación.

Criterio: Portabilidad: Se define como la característica que posee un software para ejecutarse en diferentes plataformas, el código fuente del software es capaz de reutilizarse en vez de crearse un nuevo código cuando el software pasa de una plataforma a otra. A mayor portabilidad menor es la dependencia del software con respecto a la plataforma. Se desea comparar las características de portabilidad que posea cada lenguaje de programación. La portabilidad en si no es una característica de eficiencia, pero la hemos clasificado aquí ya que mientras mas portable sea el lenguaje podría mejorar el uso del recurso tiempo al evitar recodificar un programa para distintas plataformas.

Criterio: Recolector de basura: Cualquier programa informático hace uso de una cierta cantidad de memoria de trabajo puesta a su disposición por el sistema operativo. Esta memoria tiene que ser gestionada por el propio programa. Generalmente, el programador dispone de una biblioteca de código que se encarga de estas tareas. No obstante, el propio programador es responsable de utilizar adecuadamente esta biblioteca.

Como alternativa es necesaria una gestión implícita de memoria, donde el programador no es consciente de la reserva y liberación de memoria. Esto es obligado en algunos lenguajes de programación donde no se maneja el concepto de memoria. Luego un recolector de basura es un mecanismo implícito de gestión de memoria implementado en algunos lenguajes de programación.

Criterio: Comprobación de Tipos: La comprobación de tipos es el proceso de verificar y hacer cumplir las restricciones de los tipos de datos. La comprobación de tipos se puede realizar en tiempo de compilación, lo que llamaremos comprobación estática o en tiempo de ejecución, lo que llamaremos una comprobación dinámica.

Criterio: A prueba de fallos I/O: En la mayoría de los lenguajes de programación se imprimirá un mensaje de error o una excepción si una operación de entrada y salida o alguna llamada al sistema falla (por ejemplo chmos, kill(unix)). Esto puede ocurrir a menos que el programador haya dispuesto explícitamente la manipulación de estos eventos en la codificación.

4.1.1.3 Desarrollo de restricciones.

En la modelización CSP, resulta fundamental poder disponer de un modelo de restricciones adecuado para la especificación del problema y capaz de captar las distintas tipológicas de restricciones que pueden aparecer. De acuerdo a esto definiremos algunos criterios que nos permitan medir la posibilidad de implementar distintos algoritmos para la selección de variables y de valor. Además de la posibilidad que nos presente el lenguaje para definir nuevas restricciones a partir de la ya existentes. A continuación se presentan algunos criterios que pueden ser valiosos para medir el desarrollo de restricciones en un Lenguaje de Programación con Restricciones dado. En este caso particular usará una justificación basada en una referencia bibliográfica.

Justificación Bibliográfica:

Criterio: Definición de propagadores: El objetivo de la Propagación de Restricciones es restringir el conjunto de posibles valores que pueden tomar las variables, mediante la aplicación de las restricciones, hasta que finalmente se encuentre un solo valor para cada variable. El conjunto de posibles valores se almacena en una estructura denominada depósito de restricciones (constraint store). Por ejemplo, el hecho de que una primera variable B1 debe tomar un entre 0-25, se expresa mediante la siguiente restricción: $B1 \in \{0,\dots,24\}$ en el depósito de restricciones.

Las restricciones más complejas son expresadas mediante los propagadores, los cuales inspeccionan el depósito de restricciones y lo amplían. Un propagador inspecciona el depósito respecto a un conjunto fijo de variables. Cuando se excluyen valores del dominio de una de ellas, esta acción puede añadir información sobre las otras variables en el depósito, o sea, se puede ampliar el depósito insertándole más restricciones. Como ejemplo consideremos que para la variable B1 se puede expresar instalando los siguientes tres propagadores:

B1 =<: T1 T1 =<: A1 A1 =<: S1

Para explicar cómo ellos pueden ampliar el depósito de restricciones, asumamos que

$$A1 \in \{30,\ldots,45\} \text{ y } S1 \in \{25,\ldots,60\}$$

Entonces el tercer propagador excluirá los valores 25,...,29 del dominio de S1, reduciendo su dominio a $\{30,\ldots,60\}$. Inversamente, si posteriormente se sabe que $S1 \in \{30,\ldots,40\}$, entonces A1 será restringida al nuevo dominio $A1 \in \{30,\ldots,40\}$. Nótese que este propagador permanece activo, esperando por información acerca de A1 ó S1. Sólo se detendrá cuando se tenga la certeza de que no volverá a ampliar el depósito.
El objetivo es poder comparar los lenguajes en función de la capacidad y expresividad que tienen, para poder definir estos propagadores, incorporando además funciones o clases que permitan mejorar la capacidad de implementar y utilizar estos propagadores.

Criterio: Definición de nuevos propagadores: Se busca medir la capacidad que tiene el lenguaje de programación de permitir la generación de nuevos propagadores, ya sea a través de la modificación de los propagadores actuales que posea el lenguaje como la definición desde cero de nuevos propagadores. En definitiva lo que se busca es ver la plasticidad o flexibilidad del lenguaje para realizar cambios e introducir nuevas técnicas de consistencia y reducción de dominios. Estas características se van a justificar en base a bibliografía.

Criterio: Heurísticas de selección de variables: La heurística de selección de variable es la estrategia encargada de seleccionar u ordenar las variables de forma previa a la selección del valor que se puede asignar a dicha variable. El orden en el cual las variables son asignadas durante la búsqueda puede tener un impacto significativo en el tamaño del espacio de búsqueda explorado. Esto puede ser contrastado tanto empírica como analíticamente. Generalmente las heurísticas de selección de variables tratan de utilizar lo antes posible las variables que más restringen a las demás. La intuición es tratar de asignar lo antes posible las variables más restringidas y de esa manera identificar las situaciones sin salida lo antes posible y así reducir el número de vueltas atrás. La selección de variables puede ser estática o dinámica. Se buscará entonces realizar una comparación entre los lenguajes de programación analizando cuales estregáis de ordenación de variables se utilizan por defecto o se pueden seleccionar para resolver un determinado problema.

Criterio: Generar Nuevas heurísticas de selección de variables: Se busca medir la capacidad que tiene el lenguaje de programación de permitir la generación de nuevas heurísticas de selección de variable, ya sea a través de la modificación de las heurísticas actuales que posea el lenguaje como la definición desde cero de nuevas heurísticas o ya sea la mixtura entre estas. En definitiva lo que se busca es ver la plasticidad o flexibilidad del lenguaje para realizar cambios e introducir nuevas estrategias de selección de variables. Estas características se van a justificar en base a bibliografía.

Criterio: Heurísticas de selección de valor: Las heurísticas de selección de valor tienen como objetivo seleccionar el valor más prometedor para cada variable en su dominio posible. La idea básica es seleccionar el valor de la variable actual que más probabilidad tenga de llevarnos a una solución, es decir, identificar la rama del árbol de búsqueda que sea más probable que obtenga una solución. La mayoría de las heurísticas propuestas tratan de seleccionar el valor menos restringido de la variable actual, es decir, el valor que menos reduce el número de valores útiles para las futuras variables, o alternativamente, el que deja los dominios mayores. Esto sigue la intuición de que un subproblema es más probable que tenga solución cuantos más valores tengan las variables que quedan por instanciar en sus dominios.

Criterio: Generar nuevas heurísticas de selección de valor: Se busca medir la capacidad que tiene el lenguaje de programación de permitir la generación de nuevas heurísticas de selección de valor, ya sea a través de la modificación de las heurísticas actuales que posea el lenguaje como la definición desde cero de nuevas heurísticas o ya sea la mixtura entre estas. En definitiva lo que se busca es ver la plasticidad o flexibilidad del lenguaje para realizar cambios e introducir nuevas estrategias de selección de valor. Estas características se van a justificar en base a bibliografía.

A continuación en la Tabla 4.1 se presenta a modo aclaratorio la estructuración que tienen los criterios ya vistos mediante los cuales se puede realizar el estudio de comparación entre lenguajes de programación con restricciones en función de las características internas del lenguaje.

INTERNOS	Criterios		
	Expresividad	**Eficiencia**	**Desarrollo de restricciones**
Experimental	• Líneas de código	• Tiempo de ejecución • Uso de memoria • Uso de disco • Nº backtrack	
Bibliografía	• Tipificación • Tipos de datos y estructuras • Def. de dominios • Def. de restricciones • Operaciones sobre dominios. • Integración • Sencillez	• Reutilización • Seguridad de tipos • Tipo de traducción • Portabilidad • Recolector de basura • Comprobación de Tipos • A prueba de fallos I/O	• Definición de propagadores. • Definición de nuevos propagadores • Heurísticas de selección de valor. • Generar nuevas heurísticas de selección de valor. • Heurísticas de selección de variables. • Generar Nuevas heurísticas de selección de variables

Tabla 4.1: Criterios internos de comparación

4.1.2 Criterios Externos.

Los criterios Externos buscar representar aquellas características o atributos del lenguaje que no son propios de su estructura interna. Pero sin embargo son características que se deben considerar a la hora de la selección de un lenguaje, pues son características que proveen cualidades para soportar el Lenguaje de Programación. Estos criterios simplemente dados por su característica servirán para guiar la decisión pero son difíciles de poder cuantificar. Ya que por ejemplo no se puede determinar que un lenguaje es mejor que otro por su paradigma, simplemente depende de los objetivos que tenga el avaluador en el ámbito en el cual va a utilizar el lenguaje. Otro elemento a mencionar es que por la naturaleza de estos atributos solo están sujetos a justificación bibliográfica.

Justificación Bibliográfica:

Criterio: Paradigma de programación: Un paradigma de programación provee y determina la visión y métodos de un programador en la construcción de un lenguaje. Diferentes paradigmas resultan en diferentes estilos de programación y en diferentes formas de pensar la solución de problemas. Existen múltiples paradigmas, difícilmente un lenguaje de programación pueda clasificarse solamente en un paradigma, pero si se puede determinar cual es el paradigma que predomina en su funcionamiento.

Criterio: Tipo de licencia: Una licencia de software es un contrato entre el licenciante (autor/titular de los derechos de explotación/distribuidor) y el licenciatario del programa informático (usuario consumidor /usuario profesional o empresa), para utilizar el software cumpliendo una serie de términos y condiciones establecidas dentro de sus cláusulas.

Las licencias de software pueden establecer entre otras cosas: la cesión de determinados derechos del propietario al usuario final sobre una o varias copias del programa informático, los límites en la responsabilidad por fallos, el plazo de cesión de los derechos, el ámbito geográfico de validez del contrato e incluso pueden establecer determinados compromisos del usuario final hacia el propietario, tales como la no cesión del programa a terceros o la no reinstalación del programa en equipos distintos al que se instaló originalmente.

Criterio: Integración a alto nivel: Se evalúa la capacidad del lenguaje de poder conectarse a alto nivel con modelos generados a un nivel de abstracción mayor, como el que se describe en [74]. Para poder integrar un modelo ya sea en otro lenguaje o representación para resolverlo a través de diferentes solver.

Criterio: Estándares: Según la ISO (International Organization for Standarization) la normalización es la actividad que tiene por objeto establecer, ante problemas reales o potenciales, disposiciones destinadas a usos comunes y repetidos, con el fin de obtener un nivel de ordenamiento óptimo en un contexto dado, que puede ser tecnológico, político o económico. Aquí simplemente describiremos cual es el estándar que posee o no cada lenguaje analizado.

Criterio: Grado de Penetración en el mercado: Se busca evaluar la forma en que un determinado lenguaje es capaz de abarcar diferentes áreas en las cuales se utiliza y el número de instituciones o usuarios que trabajan con este. Además de las funciones que se les han dado a estos lenguajes. En función a esto se desea realizar una comparación entre dos lenguajes de programación y como estos han cubierto las diferentes áreas de aplicación posibles.

Criterio: Soporte: El Servicio de Soporte a un lenguaje de programación comprende el apoyo a las unidades usuarias en las actividades asociadas al uso de un lenguaje. Es decir, instalación, configuración, utilización y aseguramiento de la continuidad operacional de las diferentes funcionalidades, realizando un óptimo aprovechamiento de los recursos que se posea. Esto incluye distintos medios o canales de soporte, así como la disponibilidad de los mismos.

Criterio: Finalidad de Uso: Un software o lenguaje de programación puede tener múltiples finalidades de uso. Nos referimos específicamente a la finalidad de utilización para la cual ha sido diseñado o creado el lenguaje entre estos propósitos tenemos: Web, aplicación, procesamiento de datos, uso general (ensamblador), educación, sistema de negocio, sistema distribuido, sistemas de juegos, negocios, administración, entre otros.

A continuación en la Tabla 4.2 se presenta a modo aclaratorio la estructuración que tienen los criterios ya vistos mediante los cuales se puede realizar el estudio de comparación entre Lenguajes de Programación con Restricciones en función de las características externas del lenguaje.

EXTERNOS	Criterio
Bibliografía	• Paradigma de programación • Tipo de licencia • Integración a alto nivel • Estándares • Grado de Penetración en el mercado • Soporte • Finalidad de Uso

Tabla 4.2: Criterios de evaluación Externos

4.2 Puntuación de criterios.

En la sección anterior hemos presentado una clasificación de una lista de criterios, mediante los cuales se realizará el estudio comparativo. Pero esta lista por si sola no sirve como medio cierto de comparación, ya que nos encontramos con una serie de criterios que miden características cualitativa y cuantitativa de un lenguaje. En el caso de los criterios internos, y específicamente en los que se puede comparar en función de la ejecución de una instancia específica, puede ser fácil comparar por ejemplo: el tiempo de ejecución de una instancia respecto de otra y luego discriminar en función a la diferencia de estos tiempos. Pero no es igual de fácil discriminar entre características cualitativas con repuesta booleanas, como por ejemplo si se posee o no un recolector de basura. En ese caso la repuesta solo seria afirmativa o negativa. Es por esto que en esta sección definiremos la forma de discriminar entre distinto criterios cualitativos y cuantitativos y además se deben establecer rangos para poder asignar una puntuación a cada criterio y estandarizar la puntuación entre criterios cuantitativos.

El objetivo de este trabajo es poder establecer una plantilla la cual sirva para realizar la comparación adecuada, pero que también sea un medio de comparación parametrizable, esto quiere decir que se puede modificar las ponderaciones asignadas a cada criterio de acuerdo a los intereses del evaluador y de la necesidades que este tenga y por la cual esta realizando la comparación. Esto quiere decir que la plantilla se compone de una ponderación por cada criterio además del puntaje asignado a cada criterio evaluado. Se plantea un mecanismo de puntuación para cada criterio que se encuentra en un rango de [0-100]. Donde un valor más cercano a 0 representa un bajo puntaje de evaluación y 100 es el puntaje máximo.

A continuación se explica el como se va a evaluar cada criterio en el orden en que ya fueron presentados en la sección anterior.

4.2.1 Criterios Internos.

Los criterios que se clasifican como internos podrían tener una comprobación tanto experimental como bibliográfica, pero de todas maneras se busca una manera de puntuar cada criterio independiente del carácter de este. Así se presenta una forma de puntuación para cada criterio.

4.2.1.1 Expresividad.

Criterio: Líneas de código LOC: Esta medida será obtenida tras la prueba experimental de un determinada instancia, a partir de esta medición podríamos comparar que lenguaje es capaz de sintetizar mejor el problema al implementarlo, en este sentido podríamos esperar que es mejor un lenguaje que genere menos líneas de código en contraposición a uno que se puede escribir en más líneas de código para el mismo problema. Este criterio no es por si solo capaz de determinar si el lenguaje es mas expresivo que otro, ya que al sintetizar líneas de código podría perderse capacidad de expresividad, pero al menos es un punto de partida en la comparación. Otro tema cuestionable es si es mejor un lenguaje con menos líneas de código, para este caso lo consideraremos así, esto quiere decir que a menor número de líneas de código premiaremos en puntaje nuestro lenguaje, en contraposición a otro que para la misma instancia genera más líneas de código. Por supuesto esto puede ser modificado por el evaluador de acuerdo a su criterio.

Para poder establecer un puntaje adecuado dentro del rango establecido de evaluación, necesitamos tener las LOC de cada lenguaje y calcular un valor en proporción al total de líneas de código generado en los dos lenguajes. (Bajo el supuesto de comparar dos lenguajes). Sea Loc1 y Loc2 las líneas de código del lenguaje 1 y lenguaje 2 respectivamente podemos calcular el puntaje para el lenguaje 1 como Puntaje 1 = 100 – (Loc1/(Loc1+Loc2)*100) y para el lenguaje 2 como Puntaje 2 = 100 – (Loc2/(Loc1+Loc2)*100).

Criterio: Tipificación y comprobación de tipos: Dado que la tipificación puede variar de un lenguaje a otro. Para poder puntuar la tipificación de un lenguaje se establecerá un sistema de valuación por categorías. Se supone que la tipificación de un lenguaje de programación podría clasificarse como de chequeo estático o dinámico según el momento de chequear la tipificación y también se puede clasificar entre fuertemente o débilmente tipificado según la rigurosidad para imponer las reglas de tipificación. Debemos establecer que características son mas deseables que otras y en función de eso asignar un puntaje por tipificación, nuevamente se hace referencia a que si el evaluador

desea resaltar otra característica del lenguaje puede variar la puntuación propuesta en este documento. Para este caso se premiará con mayor puntaje a aquellos lenguajes de tipificación fuerte pues mejorarán la expresividad del lenguaje y al mismo tiempo se premiara aquellos lenguajes con chequeo estático pues permiten detectar errores con anticipación a los dinámicos.

Tipificación	Fuerte	Débil
Estático	100	60
Dinámico	80	40

Tabla 4.3: Puntaje para tipificación y comprobación de tipos

Criterio: Tipos de datos y estructuras: En este caso hay que hacer un análisis paso a paso de los distintos tipos de datos y estructuras que soporta el lenguaje. Intentamos asignar una puntuación al lenguaje de acuerdo a la siguiente ponderación: Si permite el uso de tipos básicos de datos puntuamos con 20, si permite generar nuevos tipos de datos o datos abstractos asignamos 20 puntos adicionales, si maneja estructuras de datos estáticas asignamos 20 puntos más, si permite trabajar con estructuras de datos dinámicas asignamos 20 puntos adicionales. Ahora el evaluador tiene la libertad de asignar puntuaciones intermedias entre cada nivel según estime necesario.

Criterio: Definición de dominios: En este caso buscaremos poder definir tres rangos posibles de evaluación para poder asignar una puntuación al lenguaje en función de la cantidad y calidad de funciones o abstracciones para definir un dominio. Los rangos serán: *Bajo* cuando se pueden definir dominios con los tipos de datos básicos, en este caso le asignaremos una puntuación de 30. *Medio* cuando además de tener las características del nivel bajo además cuenta con tipos de datos abstractos para la definición de los dominios, en este caso asignaremos una puntuación de 50. *Alto*: Cuando además de tener todas las características de los niveles anteriores cuenta con librerías o clases especificas para trabajar con tipos de dominios para poder definir los dominios. En este caso asignaremos una puntuación de 100. Por supuesto al igual que en los otros casos el evaluador tiene la posibilidad de modificar valores entre estos rangos según estime conveniente para su evaluación.

Criterio: Definición de restricciones: En este caso buscaremos poder definir tres rangos posibles de evaluación para poder asignar una puntuación al lenguaje en función de las facilidades que este entrega para poder definir las restricciones de una variable o de múltiples variables. Los rangos serán: *Bajo* cuando se pueden definir restricciones solo con los operadores básicos para una sola variable, en este caso le asignaremos una puntuación de 30. *Medio* cuando además de tener las características del nivel bajo además cuenta con funciones que permita aplicar alguna restricción sobre grupos de variables, en este caso asignaremos una puntuación de 50. *Alto*: Cuando además de tener todas las características de los niveles anteriores cuenta con librerías o clases especificas para trabajar y definir múltiples tipos de restricciones. En este caso asignaremos una puntuación de 100. Por supuesto al igual que en los

otros casos el evaluador tiene la posibilidad de modificar valores entre estos rangos según estime conveniente para su evaluación.

Criterio: Operaciones sobre dominios: En este caso buscaremos poder definir tres rangos posibles de evaluación para poder asignar una puntuación al lenguaje en función de la cantidad y calidad de las funciones o abstracciones que permitan trabajar sobre un dominio. Los rangos serán: *Bajo* cuando se puede trabajar sobre el dominio de una variable solo haciendo uso de accesos recursivos. En este caso le asignaremos una puntuación de 30. *Medio* cuando además de tener las características del nivel bajo además cuenta con funciones que permitir trabajar sobre un conjunto de valores del dominio y además provea de funciones para trabajar sobre los dominio, en este caso asignaremos una puntuación de 50. *Alto*: Cuando además de tener todas las características de los niveles anteriores cuenta con librerías o clases especificas para trabajar sobre los dominios. En este caso asignaremos una puntuación de 100. Por supuesto al igual que en los otros casos el evaluador tiene la posibilidad de modificar valores entre estos rangos según estime conveniente para su evaluación.

Criterio: Integración: En este caso definiremos dos rangos de puntuación para el lenguaje de programación evaluado. Los niveles definidos son *Alto* y *Bajo*. Diremos que el lenguaje tiene un Bajo nivel de integración cuando solo es capaz de integrarse con otros lenguajes o medios a través de librerías compartidas. En ese caso lo puntuaremos en un rango de [0-50]. Diremos que un lenguaje tiene un Alta integración si es capaz de conectarse a través de librerías compartidas con otros lenguajes o posee la facilidad de utilizar entornos de programación implementados en lenguajes diferentes. Además debe permitir conectarse con diferentes medios gráficos. En este caso puntuaremos al lenguaje dentro del rango de [50-100] puntos. Por supuesto el evaluador tiene la capacidad de poder modificar las evaluaciones cuando dos lenguajes tengas diferencias mínimas en cuanto a su integración.

Criterio: Sencillez: Podemos clasificar a los lenguajes en dos niveles para puntuar el grado de sencillez. Definiremos los niveles *Alto* y *Bajo*. Diremos que un lenguaje tiene *Alta* sencillez cuando presente características que vuelven el lenguaje mas sencillo como la eliminación de la aritmética de punteros, si desaparecen los registros (*struct*), no se permite ni la definición de tipos (*typedef*) ni la de macros (*#define*) o permite la liberación de memoria automática. En esta situación puntuaremos al lenguaje con un valor entre [50-100]. Por otro lado podemos decir que un lenguaje presenta una Baja sencillez cuando no tiene las características antes nombradas o tiene una lógica complicada que pueda generar un menor entendimiento de cómo opera el lenguaje agregando operativas que en otros lenguajes son automáticas. En este caso podemos puntuar al lenguaje entre [0-50].

4.2.1.2 Eficiencia.

Criterio: Tiempo de ejecución: Para medir el tiempo de ejecución se utiliza una función similar a la utilizada en las líneas de código. Aunque en este caso lo que se desea es comparar en función del obtener el menor tiempo de ejecución al implementar una instancia en dos lenguajes de programación bajo estudio. Este tiempo normalmente lo

mediremos hasta obtener la primera solución en la resolución de un CSP y lo expresaremos en segundos. Para poder establecer un puntaje adecuado dentro del rango establecido de evaluación, necesitamos tener los tiempos de cada lenguaje y calcular un valor en proporción al tiempo total generado con los dos lenguajes. Sea T1 y T2 los tiempos generados con el lenguaje 1 y lenguaje 2 respectivamente podemos calcular el puntaje para el lenguaje 1 como Puntaje 1 = 100 – (T1/(T1+T2)*100) y para el lenguaje 2 como Puntaje 2 = 100 – (T2/(T1+T2)*100).

Criterio: Uso de memoria: En este caso necesitaremos comparar una instancia especifica y la luego medimos la memoria RAM utilizada por cada implementación cuando se encuentra en ejecución. Compararemos los resultados y. asignaremos un mejor puntaje a aquel lenguaje que use menor memoria, para implementar la misma instancia. Utilizamos la misma formula aplicada en la mayoría de los criterios experimentales donde: Sea M1 y M2 la memoria usada con el lenguaje 1 y lenguaje 2 respectivamente podemos calcular el puntaje para el lenguaje 1 como Puntaje 1 = 100 – (M1/(M1+M2)*100) y para el lenguaje 2 como Puntaje 2 = 100 – (M2/(M1+M2)*100).

Criterio: Uso de Disco: En este caso necesitaremos comparar una instancia especifica y la luego medimos la memoria en disco utilizada por cada implementación cuando se encuentra en ejecución. Compararemos los resultados y. asignaremos un mayor puntaje a aquel lenguaje que use menor memoria y cantidad de archivos para implementar la misma instancia. La puntuación la realizaremos de la siguiente forma. Sea D1 y D2 la memoria de disco usada con el lenguaje 1 y lenguaje 2 respectivamente podemos calcular el puntaje para el lenguaje 1 como Puntaje 1 = 100 – (D1/(D1+D2)*100) y para el lenguaje 2 como Puntaje 2 = 100 – (D2/(D1+D2)*100).

Criterio: Nº backtrack: En el caso de los backtrack hay que introducir las modificaciones necesarias para poder obtener el número de backtrack para resolver una instancia dada de CSP. Una vez obtenido esta cantidad estamos en condiciones de comparar. Suponemos que si una solución se obtiene con un menor número de backtrack es mejor que una solución en la que se incurrió en más backtrack. De esta manera si B1 y B2 es el numero de backtrack incurridos através del lenguaje 1 y lenguaje 2 respectivamente podemos calcular el puntaje para el lenguaje 1 como Puntaje 1 = 100 – (B1/(B1+B2)*100) y para el lenguaje 2 como Puntaje 2 = 100 – (B2/(B1+B2)*100).

Criterio: Reutilización: La reutilización la clasificaremos entre niveles para poder asignar a cada lenguaje de programación. Los niveles que vamos a establecer son 3 de acuerdo al grado de reutilización que se observe en el lenguaje evaluado. Estos niveles son: *Bajo*: se dirá que la reutilización es baja cuando solo permita la copia de códigos o la importación de librerías básicas compartidas, *Medio:* diremos que tiene una reutilización media cuando tenga todas las características de bajo nivel y adicionalmente cuente con la capacidad de poder importar archivos o librerías creadas por el usuario además de interfaces adecuados para modularización y comunicación entre componentes, *Alto:* definiremos un lenguaje como de alto grado de reutilización si este además de tener todas las características de los niveles anteriores, permite además la herencia, en la que se crean nuevas clases a partir de clases ya existentes absorbiendo sus atributos y comportamientos y adornándolos con capacidades que las nuevas

clases requieran. Ahora es posible asignarle a cada nivel una puntuación de la siguiente forma Bajo=30 puntos, Medio= 50 puntos, Alto=100 puntos.

Criterio: Seguridad de tipos: En este caso solo podremos identificar si el lenguaje presenta seguridad de tipos o no. Para lograr demostrar que un lenguaje tiene seguridad de tipos debe cumplir primero con Progreso: Un término bien tipificado (una “expresión”) nunca se queda atascado. Es decir, o es un valor, o podemos tomar un paso de evaluación. Y segundo debe cumplir con Preservación: Si tomamos un paso de evaluación sobre un término bien tipificado, el término sigue estando bien tipificado. Esta demostración se realiza inductivamente sobre todos los posibles términos de nuestro lenguaje. Si el lenguaje es seguro le asignamos el máximo puntaje 100 en caso contrario 0. El evaluador tiene la posibilidad se modificar la puntuación entre lenguajes seguros para definir una diferencia entre ellos. Esto requiere un análisis mayor sobre la seguridad de tipo de cada lenguaje.

Criterio: Tipo de traducción: En el proceso de traducción nos encontramos con lenguajes que usan compilador o interprete en ese caso podemos asignar una puntuación al lenguaje si es compilado o interpretado. Según los objetivos del evaluador este puede modificar la puntuación para beneficiar un lenguaje sobre otro. Siempre que se respete el rango de puntuación mínimo y máximo. El este caso particular se asignará un mayor puntaje a los lenguajes compilados sobre los interpretados, de esta forma si es compilado asignamos 100 y si es interpretado 80, solo se define una diferencia de 20 entre cada tipo para no castigar demasiado uno sobre otro.

Criterio: Portabilidad: En este caso simplemente puntuaremos al máximo al lenguaje que permita la portabilidad, mientras que asignaremos el mínimo a aquel lenguaje que no presente la posibilidad de portabilidad. Ahora si ambos lenguajes en comparación presentan portabilidad el evaluador puede discriminar según la cantidad de plataformas sobre las cuales puede correr el lenguaje. De acuerdo a lo anterior si el lenguaje es portable puntuamos 100 y en caso de no ser portable asignamos 0.

Criterio: Recolector de basura: En el caso de la recolección de basura se puede ver si existe o no un proceso de recolección de basura automático, para impedir que se generen problemas en tiempo de ejecución que pueden ser difícilmente depurábles. Aunque también tenemos que expresar que al incorporar o tener activo el recolector de basura se sacrifica tiempo de ejecución. Por lo cual el evaluador tiene la libertad de puntual al lenguaje en función si desea ganar seguridad puede puntuar con 100 al lenguaje y 0 en caso de no tener recolector de basura. Si lo que el evaluador quiere medir es en función del tiempo de ejecución puede puntuar con 100 si posee recolector, pero lo puede desactivar y 80 si no lo puede desactivar.

Criterio: A prueba de fallos I/O: En este caso simplemente definiremos si se tienen primitivas para manejar los fallos de entrada y salido o no. Es por este motivo puntuaremos con valor máximo si el lenguaje es a prueba de fallos de entrada y salido y asignaremos el mínimo en caso contrario.

4.2.1.3 Desarrollo de restricciones.

Criterio: Definición de propagadores: En este caso buscaremos poder definir tres rangos posibles de evaluación para poder asignar una puntuación al lenguaje en función de la capacidad de este para poder definir diferentes propagadores. Los rangos serán: *Bajo* cuando se pueden definir propagadores solo haciendo uso de las operaciones básicas del lenguaje, en este caso le asignaremos una puntuación de 30. *Medio* cuando además de tener las características del nivel bajo además cuenta con funciones para trabajar sobre los dominios y poder modificar los dominios de una variable, en este caso asignaremos una puntuación de 50. *Alto*: Cuando además de tener todas las características de los niveles anteriores cuenta con librerías o clases especificas para trabajar sobre los dominios. En este caso asignaremos una puntuación de 100. Por supuesto al igual que en los otros casos el evaluador tiene la posibilidad de modificar valores entre estos rangos según estime conveniente para su evaluación.

Criterio: Definición de nuevos propagadores: Se busca medir la capacidad que tiene el lenguaje de programación de permitir la generación de nuevos propagadores. En este caso usaremos una puntuación en dos rangos; *Bueno* y *Malo*. Definiremos a un lenguaje como B*ueno* cuando tenga la capacidad de poder utilizar todas las funciones de definición de propagadores y permita modificar las funciones o los propagadores que tenga el lenguaje por defecto. En ese caso le asignaremos un puntaje entre [50-100]. Definiremos un lenguaje como *Malo* cuando este presente una inflexibilidad tal que no permita realizar modificaciones a los propagadores existentes o esta flexibilidad sea demasiado restringida. En ese caso puntuaremos al lenguaje entre [0-50]. Por supuesto el evaluador tiene la posibilidad de establecer el mejor valor que estime si es que dos lenguajes tienen características similares.

Criterio: Heurísticas de selección de variables: En este caso buscaremos poder definir tres rangos posibles de evaluación para poder asignar una puntuación al lenguaje en función de las heurísticas de selección de variables que implementa. Los rangos serán *Bueno* si el lenguaje contiene funciones para implementar heurísticas de selección de variables. Las que pueden estar implementadas en librerías especificas de CP. En este caso las puntuaremos en un rango de [50-100]. *Malo* si el lenguaje presenta pocos o ninguna heurística de selección de variables. En este caso lo puntuaremos en un rango de [0-50]. Por supuesto el evaluador tiene la capacidad de poder modificar los valores dentro de los rangos establecidos según estime conveniente.

Criterio: Generar Nuevas heurísticas de selección de variables: Se busca medir la capacidad que tiene el lenguaje de programación de permitir la generación de nuevas heurísticas de selección de variable. En este caso usaremos una puntuación en dos rangos; *Bueno* y *Malo*. Definiremos a un lenguaje como B*ueno* cuando tenga la capacidad de poder utilizar todas las funciones para trabajar sobre los dominios y las variables para generar nuevas heurísticas de selección de variables, ya sea nueva o modificando las heurísticas existentes. En ese caso le asignaremos un puntaje entre [50-100]. Definiremos un lenguaje como *Malo* cuando este no tenga o impida la capacidad de poder modificar las heurísticas existentes. En ese caso puntuaremos al lenguaje entre [0-50]. Por supuesto el evaluador tiene la posibilidad de establecer el mejor valor que estime si es que dos lenguajes tienen características similares.

Criterio: Heurísticas de selección de valor: En este caso buscaremos poder definir tres rangos posibles de evaluación para poder asignar una puntuación al lenguaje en función de las heurísticas de selección de valor que implementa. Los rangos serán *Bueno* si el lenguaje contiene funciones para implementar heurísticas de selección de valor, que pueden ser a través de librerías especificas de CP. En este caso las puntuaremos en un rango de [50-100]. *Malo* si el lenguaje presenta pocos o ninguna heurística de selección de valor incorporada en el lenguaje. En este caso lo puntuaremos en un rango de [0-50]. Por supuesto el evaluador tiene la capacidad de poder modificar los valores dentro de los rangos establecidos según estime conveniente

Criterio: Generar nuevas heurísticas de selección de valor: Se busca medir la capacidad que tiene el lenguaje de programación de permitir la generación de nuevas heurísticas de selección de valor. En este caso usaremos una puntuación en dos rangos; *Bueno* y *Malo*. Definiremos a un lenguaje como B*ueno* cuando tenga la capacidad de poder utilizar todas las funciones para trabajar sobre los dominios de las variables para generar nuevas heurísticas de selección de valor, ya sea nueva o modificando las heurísticas existentes. En ese caso le asignaremos un puntaje entre [50-100]. Definiremos un lenguaje como *Malo* cuando este no tenga o impida la capacidad de poder modificar las heurísticas existentes. En ese caso puntuaremos al lenguaje entre [0–50]. Por supuesto el evaluador tiene la posibilidad de establecer el mejor valor que estime si es que dos lenguajes tienen características similares.

4.2.2 Criterios Externos.

Los criterios que se clasifican como externos solo necesitan de una comprobación bibliográfica, pero de todas maneras se busca una forma de puntuar cada criterio. Así se presenta el mecanismo de puntuación para cada criterio.

Criterio: Paradigma de programación: Dado que no podemos discriminar que un paradigma sea superior a otro a menos que sea un punto que el evaluador desee resaltar este tiene la libertad de poder puntuar libremente un paradigma sobre otro. Siempre que respete el rango de puntuación establecido. Para nuestro caso puntuaremos cualquier paradigma con un máximo de 100. Ahora la relevancia de este criterio en la comparación puede ser modificada usando un factor de importancia que se verá en las próximas secciones.

Criterio: Tipo de licencia: Una licencia de software es un contrato que entrega las reglas para utilizar o distribuir el software cumpliendo una serie de términos y condiciones establecidas dentro de sus cláusulas. Es por esto que al igual que en otros criterios cualitativos podemos puntuar a cada lenguaje de manera constante asignándole el máximo puntaje si tiene una licencia y el mínimo en caso de no tenerla. Ahora la puntuación diferenciada según el tipo de licencia queda liberado al análisis que desee el evaluador, ya que para esto se necesita algún estudio adicional que indique por que una licencia es mejor que otra.

Criterio: Integración a alto nivel: Mediremos la posibilidad que tiene el lenguaje de poder integrarse a alto nivel con otros lenguajes de modelado o tomar diferentes modelos y asimilarlos para generar una solución. En este caso definiremos simplemente la puntuación máxima (100 puntos) si este presenta la capacidad de integrarse con modelos externos a alto nivel y puntuaremos el criterio con el mínimo (0 puntos) en caso de no presentar ninguna característica deseable de integración. Como en todos los casos el evaluador puede modificar las puntuaciones según estime conveniente si es que hay dos lenguajes que presentan características de integración entonces queda a libertad del evaluador asignar las puntuaciones a estos lenguajes. Siempre que respete los rangos máximos y mínimos para las puntuaciones.

Criterio: Estándares: Si el lenguaje esta estandarizado o cumple con algún estándar lo podemos puntuar a cada lenguaje de manera constante asignándole el máximo puntaje si tiene o cumple con un estándar y el mínimo en caso de no tenerla. Ahora la puntuación diferenciada según el estándar especifico queda liberado al análisis que desee el evaluador, ya que para esto se necesita algún estudio adicional que indique por que un estándar es mejor que otro.

Criterio: Grado de Penetración en el mercado: Para el Grado de penetración en el mercado utilizaremos una clasificación por niveles en el cual definiremos un Bajo, Medio y Alto nivel de soporte. De acuerdo a esto asignaremos una puntuación a cada nivel establecido. Al bajo nivel asignaremos 20 puntos, al nivel Medio asignaremos 50 puntos y a un Alto nivel asignaremos 100 puntos. Como en todos estos casos el evaluador puede modificar las puntuación según lo estime conveniente.

Criterio: Soporte: Para el soporte utilizaremos una clasificación por niveles en el cual definiremos un Bajo, Medio y Alto nivel de soporte. De acuerdo a esto asignaremos una puntuación a cada nivel establecido. Al bajo nivel asignaremos 20 puntos, al nivel Medio asignaremos 50 puntos y a un Alto nivel asignaremos 100 puntos. Como en todos estos casos el evaluador puede modificar las puntuación según lo estime conveniente.

Criterio: Finalidad de Uso: En este caso no podemos discriminar que un lenguaje sea mejor que otro por su finalidad de uso. Sin embargo puede ser que este criterio nos ayude a decidir que lenguaje seleccionar en función de lo que el evaluador desee de él. En este caso asignaremos una puntuación máxima al criterio independiente se la finalidad de uso del lenguaje.

4.3 Ponderación de criterios.

Como ya se ha establecido en la sección anterior cada criterio tiene asignado un puntaje que esta en el rango de [0-100]. Pero como el objetivo de este trabajo es poder establecer una plantilla la cual sirva para realizar la comparación adecuada, pero que también sea un medio de comparación parametrizable. Necesitamos modificar las ponderaciones asignadas a cada criterio de acuerdo a los intereses del evaluador y de la necesidades que este tenga y por la cual esta realizando la comparación. Esto quiere decir que la plantilla se compone de una ponderación por cada criterio además del puntaje asignado a cada criterio evaluado. Se establece una división de los criterios en grupo donde se tienen los criterios internos y criterios externos como grupos iniciales. Los criterios internos a su vez se dividen en tres grupos. Los criterios para la Expresividad, Eficiencia y Manejo de restricciones como se presento en la tabla 4.1.

Cada criterio tiene una ponderación la que debe estar en el rango [0-1]. Pero cada uno de los grupos de criterios posee una determinada ponderación total la cual debe ser 1 en cada caso. Eso quiere decir que por cada grupo de criterios se debe dividir la ponderación 1 entre cada sub criterio del grupo. Esta ponderación es responsabilidad del evaluador, el cual va a asignar una mayor ponderación a los criterios los cuales le parezcan más relevantes de analizar y así irá disminuyendo la ponderación a medida que el criterio pierda importancia. Si existe la situación en la cual hay criterios que parecen completamente irrelevantes de comparar se les puede asignar ponderación 0. Con eso no se verá afectado el resto de la evaluación por este criterio. Eso si se debe tener cuidado en repartir las ponderaciones para que al final el grupo de criterios tenga la ponderación máxima 1. A continuación se presenta la tabla 4.4 que representa una posible asignación de ponderación a los diferentes criterios.

	Puntaje [0-100]	Ponderación [0-1]	Total [0-100]
CRITERIOS INTERNOS			
Expresividad			
Líneas de código	0	0,2	0
Tipificación	0	0,1	0
Tipos de datos y estructuras	0	0,1	0
Def de dominios	0	0,1	0
Def de restricciones	0	0	0
Operaciones sobre dominio	0	0,3	0
Integración	0	0,15	0
Sencillez	0	0,15	0
SUB TOTAL		1	0

Tabla 4.4: Fragmento de la tabla de puntuación y ponderación

Una vez obtenidos los puntajes de cada criterio y las ponderaciones asignadas a cada criterio se procede a realizar la siguiente operación Puntaje criterio * Ponderación criterio = Puntaje total criterio. Luego se debe sumar todos los puntajes totales por criterio y obtener el Subtotal del grupo de criterio. Análogamente con este cálculo se tendrá la puntuación de cada grupo de criterios.

Adicionalmente se puede ponderar la importancia de cada grupo de criterio en un rango de [0-1] en función de la importancia del grupo dentro de la evaluación que se este realizando, por ejemplo si quisiéramos comparar los lenguajes de programación en función principalmente de la eficiencia, bastaría con aumentar la ponderación de este grupo de criterios en comparación con los otros. Igualmente si existe un grupo de nula relevancia para la evaluación se puede ponderar con 0. Así de manera similar a como se realizo el calculo de puntaje de cada criterio, se realiza el calculo de puntaje de los grupos de criterios en función de cada subtotal. O sea se debe multiplicar el subtotal de cada grupo por su factor, lo cual entregaría el puntaje total de evaluación del lenguaje. Esto puede verse reflejado en la Tabla 4.5.

Grupos	Sub totales	Ponderación	Total
Expresividad	0	0,2	0
Eficiencia	0	0,4	0
Desarrollo de restricciones	0	0,3	0
CRITERIOS EXTERNOS	0	0,1	0
TOTAL EVALUACION		1	0

Tabla 4.5: Ponderación de los grupos de evaluación

Como ya hemos visto se puede realizar una evaluación a un lenguaje, en la cual se puede entregar una puntuación por cada criterio, pero podemos cambiar las ponderaciones de cada criterio o grupo de criterios dando a la evaluación una característica parametrizable. De esta manera a partir de una puntuación inicial, se podrían modificar las ponderaciones y así comparar en función de distintos puntos a él o los lenguajes que se estén evaluando.

CAPÍTULO 5

Prueba de la Plantilla de Comparación

5 Prueba de la Plantilla de Comparación.

Hasta este punto solo hemos definido cuales son los criterios que se utilizarán para comparar los Lenguajes de Programación con Restricciones, además de establecer una plantilla de comparación en función de la asignación de puntajes y ponderación de los criterios, para comparar de manera cuantitativa los lenguajes.

En esta sección nos centraremos en realizar una comparación entre dos lenguajes de programación, para probar de manera empírica el funcionamiento de los criterios y de la plantilla. Para esto desarrollaremos un Caso de Comparación, el cual consiste en seleccionar dos lenguajes de programación que se compararán haciendo uso de la Plantilla de Comparación. Este Caso no es nada más que una situación particular de evaluación para utilizar la propuesta de comparación presentada en la sección 4.

Posteriormente se realiza la comparación en función de los criterios que se pueden justificar solo con una investigación bibliográfica. Luego se analizan los problemas que se pueden resolver y se seleccionan dos modelos para implementar en los Lenguajes de Programación seleccionados. Luego se realizan las pruebas para comparar los criterios que necesitan de una comprobación experimental. Finalmente se analizan los puntajes obtenidos en este Caso de valuación y se presentan los resultados.

5.1 Selección de Lenguajes para la comparación.

Se realizará una comparación imparcial de dos solvers de FD, primero se selecciona un Lenguaje de Programación Lógico y posteriormente un Lenguajes Orientados a Objetos de los nombrados en las secciones previas. Por parte de los Lenguajes Lógicos se realizará el estudio con Eclipse. Esto por su popularidad dentro de los grupos de solver. Por parte de los Lenguajes Orientados a Objetos se selecciona GeCode, para realizar el estudio comparativo. Se selecciona principalmente estos lenguajes dado su popularidad en la comunidad de CP, además que estos lenguajes han sido utilizados en varia otras investigaciones [74], son de acceso libre y presentan un amplio soporte en documentación para trabajar con ellos. Otro factor importante es que son algunos de los solver que presentan más facilidad y simplicidad para la implementación y modificación en cada uno de sus respectivos paradigmas [83] [84].

Aunque los distintos sistemas con restricciones han sido probados para solucionar un gran número de Modelos tradicionales, el trabajo más comparativo ha sido hecho por los propios autores de estos lenguajes. La idea es que este estudio sea imparcial para comparar los distintos lenguajes planteados. Se espera que este estudio pueda guiar a los desarrolladores a implementar las soluciones a los problemas específicos de satisfacción de restricciones.

Para realizar las pruebas experimentales con cada lenguaje se realizará un estudio previo para justificar el uso de cada problema o modelo. Ya que tenemos los lenguajes que se van a comparar podemos iniciar el proceso de comparación inicialmente teórico y posteriormente experimental.

5.2 Comparación de criterios bibliográficos para el Caso: Eclipse y GeCode/J.

A Continuación se realizará un pequeño análisis comparativo entre las características de cada lenguaje respecto a los puntos de comparación sujetos a comprobación bibliográfica (teórica) definidos en la sección anterior. Con esto se busca definir cuales son las diferencias o similitudes de los lenguajes, revisando en cada caso como cumplen con los criterios antes definidos. No se diferenciará a los criterios en grupos, solo se agrupan por su carácter bibliográfico, ya que esta clasificación se expresará a la hora de puntuar el caso en la plantilla de evaluación.

Criterio 1: Tipificación

Eclipse: Dado la rigurosidad con la que eclipse implementa la tipificación lo podemos clasificar como fuertemente tipificado y es capaz de realizar este análisis en tiempo de ejecución por lo cual diremos que es un lenguaje de comprobación de tipos dinámico [11].

GeCode: Por estar basado en C++ y específicamente en el caso de GeCode/j el cual esta basado en java, son lenguajes fuertemente tipificados. El cual presenta una comprobación de los tipos de datos tanto estático ósea cuando compila y dinámico cuando esta en tiempo de ejecución [73].

Criterio 2: Tipos de datos y estructuras

Eclipse: Soporte datos numéricos como enteros, flotantes, racionales y reales acotados, además es capaz de soportar cadenas de caracteres entre comillas dobles. Soporta trabajo con átomos ósea con constantes simbólicos, además de listas que son construidas a partir de una cabeza y colas que pueden ser otras listas de manera recursiva. En cuanto a las estructuras (funciones) tienen un nombre y una determinada aridad ósea el número de argumentos que soporta.

GeCode: Soporta tipos de datos numéricos como enteros, flotantes, doubles, boléanos. Además soporta datos de tipo punteros, caracteres y cadenas de caracteres. Bajo el interfaz java se puede soportar datos de tipo String. Otra facilidad es que se pueden definir tipos de datos de cualquier estructura que se defina (definición de estructuras). En las estructuras de datos soporta arreglos unidimensionales y multidimensionales [61].

Criterio 3: Definición de dominios

Eclipse: Tiene un amplio soporte a dominios finitos haciendo uso de la librería *ic* (interval solver) esta librería implementa dominios finitos de números enteros, junto con un conjunto básico de restricciones. Además posee la librería *ic_symbolic* el cual permite trabajar con dominio simbólicos (por ejemplo nombres de productos). Junto con la librería *ic* se puede utilizar la *ic_sets* para manejar restricciones sobre dominios de conjuntos finitos de enteros.

GeCode: Una de las principales librerías que se utilizan para definir los dominios es la *gecode/set.hh* la cual contiene una serie de clases para trabajar sobre la definición de conjuntos de variables y de los dominios de las variables. Donde se pueden implementar dominios como intervalos o como arreglo de datos. Para definir dominio se utiliza el la funcion *Gecode::dom*(). También se puede hacer uso de *SetVar()* para definir conjuntos [11].

Criterio 4: Definición de restricciones

Eclipse: Cuenta con suspend la cual permite suspender una restricción hasta que las variables son instanciadas y luego las prueba. También hace uso de la librería *ic* para restricciones. Además cuenta con *ic_cumulative*, *ic_edge_finder* , las cuales permiten restricciones globales para aplicaciones de scheduling. Se tiene *ic_sets* que implementa las restricciones sobre el dominio de los conjuntos finitos de números enteros. [75], se pueden definir restricciones boleanas y lineales. Eclipse proporciona una sintaxis común para las restricciones de la aritmética principal provista por diferentes solvers. La idea básica es que el nombre y la sintaxis de la restricción determine el sentido declarativo, mientras que la semántica operacional está determinada por el módulo que implementa la restricción.

GeCode: Se pueden definir restricciones de relaciones simples sobre variables enteras cumpliendo relaciones entre variables enteras y valores enteros. La relación depende del tipo de relación entera *IntRelType,* la que se puede definir como *rel(home, x, <IntRelType>, y),* donde *<IntRelType>* define el tipo de relación entre variables enteras. Se pueden definir de la misma manera restricciones de relaciones simples sobre variables booleanas, de la misma manera anterior solo que ahora se usa un tipo *BoolOpType,* la que se puede definir como *rel(home, x, < BoolOpType >, y),* donde *< BoolOpType >* define el tipo de relación entre variables booleanas. También se pueden definir restricciones aritméticas sobre variables enteras y restricciones lineales usando la función *linear().* A continuacion se presenta una lista de restricciones globales que se pueden usar:.*all_different, among, at_least, at_most, count, cumulative, disjoint, distribute, exactly, increasing, inverse, lex_lesseq, lex_less, maximum, minimum, nvalue, partition_set, regular, sort, sum_pred.* [77].

Criterio 5: Operaciones sobre dominios.

Eclipse: Cuenta con librerías como *ic_global* la cual soporta una variedad de restricciones, lo cual requiere como argumento una lista de variables de dominio finito, de longitud no especificada. Algunas de ellas son *alldifferent/1, maxlist/2, occurrences/3* y *sorted/2*.

GeCode: Cuenta con una seria de librerías las cuales permiten trabajar sobre los dominios de las variables, para aplicar funciones sobre una o un conjunto de ellas como ocurre con *distinct()* conocida como *alldifferent*. Cuenta con una serie de funciones las que permiten obtener distinto valores y realizar distintas operaciones sobre los dominios. Entre ellas podeos considerar *cardinality()* que puede acotar el dominio de una variables entre dos valores [77].

Criterio 6: Integración

Eclipse: Cuenta con la librería *Eplex* la cual permite conectarse con solver externos para obtener otras soluciones usando solvers de programación entera XPRESS o CPLEX [75] tiene la deficiencia que requiere de la licencia de un software externo. Otra librería es la *clpqr* la cual ofrece dos implementaciones del método Simplex para la resolución de restricciones lineales. Esta biblioteca cuenta con software de dominio público, y puede ser usado para los problemas pequeños (con menos de 100 variables). Eclipse posee librerías para conectarse a nivel de interfaces gráficas y posee librerías con interfaces para trabajar con bases de datos Oracle y mySQL

GeCode: Posee múltiples interfaces para conectarse con distintos lenguajes de programación, Ya se conoce las librerías de Gecode/J, la cual cuenta con sus propios propagadores, motores de búsqueda y ramificación, pero son los mismos que en Gecode. Además se cuenta con la capacidad de trabajar con FlatZinc. También se cuenta con un marco de trabajo con restricciones para trabajar con Haskell. Tiene un interfaz para resolver problemas en Ruby con Gecode. También cuenta con GECOL para trabajar con Common Lisp y el proyecto GeOz que busca integrar GeCode con Mozart.

Criterio 7: Sencillez

Eclipse: Eclipse presenta una sintaxis que es una mezcla de distintos lenguajes de programación lógico, por supuesto todos usan como base Prolog. Los lenguajes lógicos como Eclipse presentan una gran representatividad para traducir los modelos a la codificación, esto genera una sencillez en la traducción del modelo y posteriormente en el uso de los métodos de resolución. Ahora bien el proceso de aprendizajes de un lenguaje lógico como Eclipse toma algún tiempo adicional, ya que para trabajar con lenguajes lógicos se necesita un conocimiento adicional de la recursividad y la declaración de hechos y predicados, para dar solución a un problema específico.

GeCode: La característica de sencillez de GeCode esta dada por la facilidad de poder trabajar con un lenguaje basado en C++ y en Java o cualquier otro interfaz que se dese utilizar. La masificación de estos lenguajes facilitan el

proceso de aprendizaje. Además cuenta con una seria de clases ya definidas para el trabajo especifico de programación con restricciones. Ahora bien comenzar a implementar utilizando GeCode es un tanto dificultoso, dado la cantidad de clases que tiene para realizar las implementaciones necesarias. La sencillez de su modelado y codificación es la misma que encontramos en lenguajes como C++. En definitiva si utilizamos Gecode/j nos encontramos con una gran sencillez, ya que se evitan una serie dificultades como eliminación de *struct*, typedef, recolectores de basura automático, etc.

Criterio 8: Reutilización

Eclipse: En este caso no proporciona una reutilización en base a importar código o componentes directamente, pero si se puede hacer referencia a librerías previamente implementadas, pero tiene que ser absorbidas por el lenguaje. La forma tradicional de reutilizar código es escribiendo en el mismo código fuente las funciones que se deseen incorporar en el programa, lo cual extiende enormemente el código escrito. Se puede implementar la reutilización de predicados haciendo uso de *:- use_module* para llamara los módulos creados [76].

GeCode: Para reutilizar el código en Gecode creamos nuevas clases pero, en lugar de partir de cero, partimos de clases, relacionadas con nuestra clase, que han sido ya creadas y depuradas. Una forma de hacer esto es crear objetos de nuestras clases existentes dentro de la nueva clase. Esto se conoce como composición porque la nueva clase está compuesta de objetos de clases existentes. Otra forma es crear una nueva clase como un tipo de una clase ya existente. Tomamos la forma de la clase existente y añadimos código a la nueva, sin modificar la clase existente. Esta forma de crear nuevos objetos se llamada herencia, y lo que hacemos es extender la clase en la que nos basamos para crear la nueva [61].

Criterio 9: Seguridad de tipos

Eclipse: Eclipse es capaz de detectar en tiempo de ejecución los errores de tipo que se les pueden dar como paso de argumentos a un predicado, en caso de asignar tipos de datos erróneos al tipo de datos asignado como parámetro al predicado al cual se a invocado [11].

GeCode: En el caso de usar el core de GeCode en c++ se cuenta el lenguaje es poco seguro referente al análisis de los tipos. Pero trabajando sobre Gecode/j se tiene un sistema seguro en el análisis de los tipos de datos, en el caso de GeCode cuenta con un análisis estático, pero el en caso de GeCode/j el análisis también se realiza en tiempo de ejecución, en ese caso diremos que realiza un análisis dinámico.

Criterio 10: Tipo de traducción

Eclipse: Es un lenguaje compilado. Se construye alrededor de un compilador incremental, que compila código fuente ECLIPSE en código WAM [72]. Realiza un proceso de compilación, el código compilado se va cargando inmediatamente en memoria.

GeCode: Puede ser compilado con modernos compiladores de C++ y se ejecuta en una amplia gama de plataformas, lo que además le da una gran portabilidad. En el caso de Gecode/j de compila con bajo una maquina virtual java compatible con el sistema operativo el cual se este utilizando. En este caso se compila, pero luego el bytecode es interpretado.

Criterio 11: Portabilidad

Eclipse: En este caso solo proporciona cierta portabilidad, pero de solver a través del uso de la librería *Eplex*. Otra característica de portabilidad puede ser cuando permite la precompilación de archivos ECO, que es un archivo objeto, que permite acelerar el proceso de compilación. Pero solo debe ser ejecutado bajo la misma versión de compilador.[11]. Posee soporte a las siguientes arquitecturas Linux/x86, Linux/x86-64, Sparc/Solaris, Mac OS X/Intel 32-bit, Mac OS X/Power PC 32-bit, Windows NT (4.0 or newer), 2000, 2003, XP, Vista 32-bit [78].

GeCode: Trabajando con Gecode se puede compilar bajo varias plataformas, pero eso no quiere decir que sea portable, cuando se utiliza Gecode/j se puede generar una precompilación y se obtiene el bytecode, un archivo que puede ser compilado bajo cualquier plataforma haciendo uso del la maquina virtual de java adecuada para el sistema operativo [referencia manual gecode]. Existen verciones de gecode para Windows x86 , Windows x64, Mac OS X ≥ 10.4 (Intel only, 32 and 64 bit), Unix y Linux.

Criterio 12: Recolector de basura

Eclipse: Proporciona un recolector de basura para la pila global, y un recolector de basura de diccionario, que se llama cuando una cierta cantidad de nuevo espacio ha sido consumido. Ya que algunos programas pueden hacer crecer la pila global cuando crean estructuras de datos o hay átomos sin utilizar. Estos recolectores funcionan automáticamente, sin embargo el uso del recolector de basura disminuye la eficiencia en tiempo del compilador de Eclipse, por lo cual se tienen predicados para controlar su funcionamiento [11] .

GeCode: En GeCode se utilizan punteros, reservas de memoria (con las ordenes malloc, new, free, delete...) y otra serie de elementos que dan lugar a graves errores en tiempo de ejecución difícilmente depurábles. Si trabajamos con Gecode/j se tienen operadores nuevos para reservar memoria para los objetos, pero no existe ninguna función explícita para liberarla. La recolección de basura es una parte integral de Java durante la ejecución de sus programas. Una vez que se ha almacenado un objeto en el tiempo de ejecución, el sistema hace un seguimiento del estado del objeto, y en el momento en que se detecta que no se va a volver a utilizar ese objeto, el sistema vacía ese espacio de memoria para un uso futuro. Esto facilita tanto la sencillez en la programación como la gestión de memoria, pero probablemente no la eficiencia [77].

Criterio 13: A prueba de fallos I/O

Eclipse: Las entradas y salidas en Eclipse se realizan a través de los canales de comunicación llamado streams. Por lo general son asociados con un archivo, un terminal, un socket, una tubería, o en colas en memoria y en buffers. Los streams se pueden abrir sólo de entrada, sólo de salida, o para la entrada y de salida. Cuando alguno de estos procesos falla se arroja un error de I/O. esto se reporta a través del predicado de stream *stderr*, que es un stream de error estándar [11].

GeCode: Nuevamente nos encontramos con una forma para GeCode y otra para GeCode/j, en este caso Gecode no presenta un modo seguro de entrada y salida, sin embargo en el caso de Gecode/J dado que esta basado en java cuenta con la capacidad de controlar los streams de salida y entrada, para poder detectar errores de entrada y salida, lo cuales son reportados en una excepción [61].

Criterio 14: Definición de propagadores.

Eclipse: Se usa la librería *repair* la cual permite marcar un valor tentativo el cual no puede se instanciado con ninguna variable. Este valor tentativo puede violar las restricciones de una variable, en cuyo caso se registra la restricción en una lista de restricciones violadas. La biblioteca *repair* también es compatible con los invariantes de propagación [11]. El uso de invariantes sirve cuando el valor de una variable cambia y consecuencia del cambio se pueden propagar a otras variables cuyos valores dependen de la variable modificada.

GeCode: Gecode proporciona múltiples propagadores con diferentes niveles de consistencia, todas las funciones toman un argumento opcional de tipo *IntConLevel*, para controlar que propagador es elegido para una restricción particular, estos tipos pueden ser: propagación por valor (*naive*), propagación por los limites o consistencia de limites, propagación por dominio o consistencia de dominio,.Adema cuenta con las librerias Gecode/set.hh mediante la cual podemos usar la función propagate(), para implemenar propagadores. Donde se puede usar una serie de relaciones entre variables *all_different, among, at_least, at_most, count, cumulative, disjoint, distribute, exactly, increasing, inverse, lex_lesseq, lex_less, maximum, minimum, nvalue, partition_set, regular, sort* [77].

Criterio 15: Definición de nuevos propagadores

Eclipse: Se cuenta con la librería *propia* la cual permite tomar cualquier predicado y modificarlo para crear nuevos propagadores. También se puede hacer uso de CHR las reglas de manejo de restricciones, las cuales se pueden usar para junto con la librería *suspend* para controlar las restricciones y por ende en trabajo con propagadores.

GeCode: GeCode cuenta con la capacidad de implementar propagadores simple para las restricciones simple a través de variables de tipo entero. Aunque se puede usar la misma forma de realizar propagadores para variables enteras, para escribir propagadores para otros dominios de variables. La modificación de funciones en Gecode es un

poco difícil, cuenta con muchos modificadores para los propagadores, pero no se tiene una plasticidad para modificar o crear nuevos. Se podrían modificar las clases escritas pero hay que tener especial cuidado con mantener las interfaces de la clase.

Criterio 16: Heurísticas de selección de variables.

Eclipse: En Eclipse usando la librería *ic* se puede aplicar el predicado *labeling/1* sobre al cual le pasa como parámetro la lista de variables las cuales van a ser instanciadas usando *indomain/1* usando técnicas de búsqueda completas. Cuenta además con la librería *ic_search* la cual contiene el predicado *search/6,* la cual implementa diversas variantes de un árbol de búsqueda incompleta. s*earch/6* puede usar distintas heurísticas como: First Solution, Bounded Backtrack Search, Depth Bounded Search, Credit Search, Limited Discrepancy Search [75].

GeCode: En GeCode se hace uso de estas heurísticas al momento de realizar las búsquedas o ramificaciones para esto se utiliza la librería *gecode/branch.hh* donde se aplica la función *branch(home, x, <IntVarBranch>,<IntValBranch>,)* donde *IntVarBranch* especifica el tipo de selección de variables a usar. A continuación se muestran algunas estrategias para variables enteras en la tabla 5.1.

INT_VAR_NONE	Primer lugar no se asigna
INT_VAR_RND	Randomico
INT_VAR_DEGREE_MIN	Con menor grado
INT_VAR_DEGREE_MAX	Con mayor grado
INT_VAR_AFC_MIN	Con menor numero acumulado de fallos.
INT_VAR_AFC_MAX	Con mayor numero acumulado de fallos
INT_VAR_MIN_MIN	Con el min más pequeño.
INT_VAR_MIN_MAX	Con el min más grande.
INT_VAR_MAX_MIN	Con el max más pequeño.
INT_VAR_MAX_MAX	Con el max más grande.
INT_VAR_SIZE_MIN	Con el tamaño más pequeño del dominio.
INT_VAR_SIZE_MAX	Con el tamaño más grande del dominio

Tabla 5.1: Estrategias de selección de variables entera

Criterio 17: Generar Nuevas heurísticas de selección de variables

Eclipse: En eclipse se tienen múltiples predicados para la manipulación de listas y específicamente para trabar sobre un conjunto de variables a ser seleccionada. Estas funcionalidades se encuentran incluidas en las librerías *ic, ic_search, list*. Donde existe una gran libertad para poder modificar el proceso del predicado *labeling* y cambiar la estrategia para seleccionar las variables antes de realizar la asignación de valor con *indomain/1*ó *indomain/2.* Donde podemos cambiar la estrategia *naive* que viene pre-definido en la estrategia de *labeling*, se puede modificar para usar

la estrategia *firt-fail* para tomar la variable con menor dominio. Para eso se usa el predicado *delete/5* que se encuentra en la librería *ic_search.* Así haciendo uso de las técnicas de programación y los predicados de las librerías para manipulación de listas se pueden ir modificando las estrategias de *labeling* para incorporar nuevas estrategias de selección de variable [75].

GeCode: En Gecode se utiliza la librería *gecode/branch.hh* para disponer de una serie de clases en las cuales se puede utilizar la función *branch(home,x,<IntVarBranch>,INT_VAL_MIN),* la cual sirve para realizar las ramificaciones se puede indicarla estrategia de selección de variable con el valor *IntVarBranch*. El problema principal radica en que estas estrategias están ya implementadas. Se podrían crear nuevas estrategias de selección de variables, pero para eso hay que reprogramar las clases ya implementadas. Hay una dificultad inherente dada por a dependencia entre clases.

Criterio 18: Heurísticas de selección de valor.

Eclipse: En Eclipse usando la librería *ic* se puede aplicar el predicado *labeling/1* sobre al cual le pasa como parámetro la lista de variables las cuales van a ser instanciadas usando *indomain/1* la cual asigna un valor a la variable dentro del dominio. Cuenta además con la librería *ic_search* se puede usar no solo *indomain/1* la cual aplica la técnica de selección de variable estática, ósea en el orden del dominio, sino que también se puede usar *indomain/2,* la cual puede agregarle mayor información al proceso de selección de variable por ejemplo usando *middle* para seleccionar un valor central del dominio [75].

GeCode: En Gecode se hace uso de estas heurísticas al momento de realizar las búsquedas o ramificaciones. Para esto se utiliza la librería *gecode/branch.h*, la cual contiene una serie de clases para poder seleccionar el tipo de estrategia de selección de valor. Se utiliza el modificador *ValSel,* donde se pueden elegir estregáis como: *ValMin*, para seleccionar el valor mínimo.*ValMed*, para seleccionar un valor medio. *ValRnd,* para seleccionar un valor de forma randómica. Tambien se tienen otras como *ValSplitMin*, *ValRangeMin*, *ValZeroOne*. Para seleccionar el valor se aplica la función *branch(home, x, <IntVarBranch>,< IntValBranch >)* donde *IntValBranch* especifica el tipo de selección de valor. Algunas se muestran en la tabla 5.2 [77].

INT_VAL_MIN	Selecciona el valor más pequeño
INT_VAL_MED	Selecciona un valor medio
INT_VAL_MAX	Selecciona el valor mas grande
INT_VAL_RND	Secciona un valor randómico
INT_VAL_SPLIT_MIN	Selecciona un valor menor al promedio del mayor y menor.
INT_VAL_SPLIT_MAX	Selecciona un valor mayor al promedio del mayor y el menor.
INT_VAL_RANGE_MIN	Seleciona el rango más pequeño del dominio de la variable.
INT_VAL_RANGE_MAX	Seleciona el rango más grande del dominio de la variable.
INT_VALUES_MIN	Trata todos los valores desde el más pequeño.
INT_VALUES_MAX	Trata todos los valores desde el más grande

Tabla 5.2: Estrategias para selección de valor

Criterio 19: Generar nuevas heurísticas de selección de valor.

Eclipse: En eclipse se tienen múltiples predicados para la manipulación de listas y específicamente para trabar sobre los dominios de las variables. Estas funcionalidades se encuentran incluidas en las librerías *ic, ic_search, list*. Donde existe una gran libertad para poder modificar el proceso del predicado *labeling* y cambiar la estrategia para seleccionar las variables cuando se realiza el *indomain/2* donde se puede usar la estrategia *middle* para la selección de valor. [75]

GeCode: En Gecode se utiliza la librería *gecode/branch.hh* para disponer de una serie de clases en las cuales se puede utilizar la función *branch(home,x,<IntVarBranch>, <IntVarBranch>),* la cual sirve para realizar las ramificaciones se puede indicarla estrategia de selección de valor con el modificador *IntVarBranch*. El problema principal radica en que estas estrategias están ya implementadas. Se podrían crear nuevas estrategias de selección de valor, pero para eso hay que reprogramar las clases ya implementadas. Hay una dificultad inherente dada por a dependencia entre clases.

Criterio 20: Paradigma de programación

Eclipse: Pertenece al paradigma de programación lógico, específicamente a los lenguajes de programación declarativa lógica. [78] Dado que este lenguaje hereda sus características del lenguaje Prolog y de algunas de sus variaciones las cuales fueron comentadas en secciones previas.

GeCode: Pertenece al paradigma de lenguajes de programación Orientados a Objeto, aunque esta orientado específicamente a resolver problemas de programación con restricciones. Esta basado principalmente en paradigmas imperativos procedural, orientado a objeto y reflexivo [77].

Criterio 21: Tipo de licencia

Eclipse: Es un lenguaje de código abierto por parte de Cisco en un estilo de licencia publica *Mozilla-License 1.1* en septiembre de 2006. La fuente oficial de repositorios y otros recursos de desarrollo están en http://www.sourceforge.net/projects/eclipse-clp. El primer lanzamiento fue a partir de la versión de código abierto 5.10 [78].

GeCode: El software y su documentación son propiedad de los autores que figuran en cada archivo. Todos los archivos, a menos que expresamente se advierta en algún archivo individual, podrán utilizarse de acuerdo a los términos de la licencia MIT. [61] La licencia MIT es una de tantas licencias de software que ha empleado el MIT (Massachusetts Institute of Technology). El texto de la licencia no tiene copyright, lo que permite su modificación. No obstante esto, puede no ser recomendable e, incluso, muchas veces dentro del Open Source desaconsejan el uso de este texto para una licencia, a no ser que se indique que es una modificación, y no la versión original [79].

Criterio 22: Integración a alto nivel

Eclipse: Eclipse presenta una arquitectura, la cual puede integrarse con interfaces graficas de java, pero a pesar de eso no es capaz de tomar un modelo y poder interpretarlo como un conjunto de cláusulas y restricciones para resolverlo. Algunos aproximaciones a este tipo de modelo se presentan en [74]. Pero no es una característica propia del lenguaje.

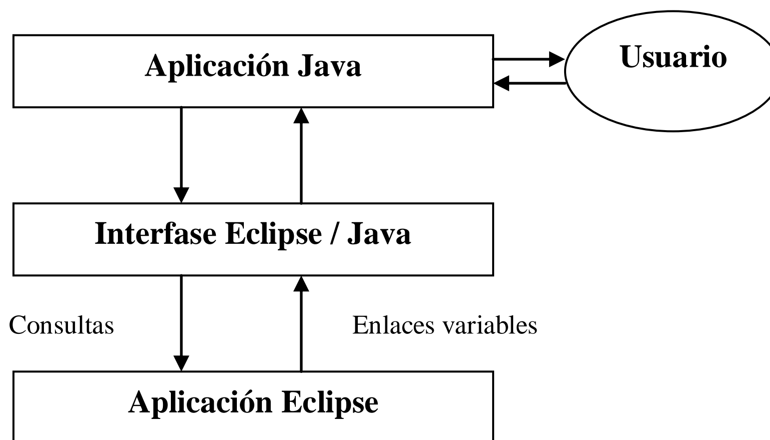

Figura 5.1: Estructura global de aplicación de sistemas desarrollado por Parc Technologies para Eclipse

GeCode: GeCode utiliza una técnica llamada de auto-enlace con Visual Studio el cual permite integrar Visual Studio para compilar usando las dlls necesarias para trabajar de manera integrada con GeCode. Pero esto no es mas que la integración con otro Interfaz para trabajar con este lenguaje. En párrafos anteriores hemos hecho referencia a

los múltiples proyectos para generar la integración entre GeCode y otros lenguajes, pero ninguno de estos incorpora una forma de poder traspasar los modelos en un alto grado de abstracción a el solver de GeCode. Solo existen algunos acercamientos para tomar las características de fácil modelado de otros lenguajes para integrarlos con el solver Gecode [77]. Otros intentos por realizar la integración de este solver con otros modelos a un nivel mayor de abstracción se presentan en [74].

Criterio 23: Estándares

Eclipse: Al ser un lenguaje basado en Prolog cumple con el estándar ISO/IEC 13211-1 en 1995, llamado ISO-Prolog . No cuenta con un estándar propio para el lenguaje.

GeCode: GeCode cumple el estándar de C++ en cual es un estándar ISO/IEC 14882 desde 1998. En el caso de Gecode/J cumple con los estándares que cumple Java los cuales son los que adopto Sun Microsystems en el 2007 el cual es GNU General Public License (GPL).

Criterio 24: Grado de Penetración en el mercado

Eclipse: Eclipse cuenta inicialmente con un gran respaldo en el mercado al ser adoptado por Cisco. Donde se utiliza para investigar la resolución de resolver problemas de trabajo en redes, pero inicialmente a sido utilizado por instituciones como ECRC (European Computer Res Centre) , IC-Parc (Imperial College London), Parc Technologies Ltd, Londres, La comunidad de código abierto que trabaja bajo la licencia estilo-Mozilla. Además hay múltiple universidades que lo incorporar como lenguaje para la investigación en temas de CP y optimización para problemas combinatoriales.

GeCode: GeCode cuenta con un amplio apoyo en distintos proyectos de desarrollo para GeCode entre ellos podemos encontrar: CP(Graph) y CP(Map) del departamento de ingeniería e informática de Universidad Católica de Louvain, de Bélgica. Qecode de la universidad de Orléans, de Francia. TuLiPa de Eberhard Karls Universität Tübingen, de Alemania. CPSP Por el Grupo de Bioinformática, Albert Ludwigs University Freiburg, de Alemania. CP4IM, Declarative Languages & Artificial Intelligence Group, K.U.Leuven, de Belgica.

Criterio 25: Soporte

Eclipse: Posee una amplio soporte, inicialmente podemos acceder a el website [78] donde podemos encontrar documentación: la cual incluye tutoriales, manuales de usuario, manual de librerías, manual de referencias, guía para desarrollo de aplicaciones, herramientas de interfaz, ejemplos, reporte de errores y contactos vía email. Notas: las cuales tienen todo tipo de información, actualizaciones etc. Registro de cambios. Planes de trabajo actual y futuro, como aportar a Eclipse y contribuciones, donde se puede encontrar información de todos los grupos participantes del proyecto, y preguntas frecuentes. Todo esto sin considerar la cantidad de tutoriales y manuales existentes en la red.

GeCode: Podemos encontrar una gran documentación en el sitio web de GeCode y lo mismo ocurre en el sitio de GeCode/j la cual incluye manuales de usuario, manual de librerías, clases, herramientas de interfaz, código escrito, reporte de errores y mecanismos de contactos, registro de cambios, Se puede encontrar información de todos los grupos participantes del proyecto y proyectos paralelos o complementarios. Además se cuentan con un listado de publicaciones sobre GeCode [61].

Criterio 26: Finalidad de Uso

Eclipse: Es un sistema de software de código abierto para el desarrollo rentable y el despliegue de aplicaciones de programación con restricciones, por ejemplo, en las áreas de planificación, programación, asignación de recursos, horarios, transporte, etc. También es ideal para la enseñanza de casi todos los aspectos del problema de combinatoria de problemas, por ejemplo, modelización de problema, programación con restricciones, programación matemática, y las técnicas de búsqueda [78].

GeCode: Es un entorno abierto, libre, portátil, accesible y eficiente para el desarrollo de sistemas basados en restricciones y aplicaciones. Orientado para resolver problemas de programación, planificación, etc. Gecode también es una herramienta de educación para quienes trabajan en el área de programación con restricciones [61].

5.3 Comparación de criterios prácticos para el Caso: Eclipse y GeCode/J.

A Continuación se realizará un pequeño análisis comparativo entre las características sujetas a análisis experimental de cada lenguaje respecto a los puntos de comparación definidos en la sección anterior. Para Eclipse se considera las librerías estándar, a pesar de poder generar múltiples utilidades adicionales a partir de estas librerías [11]. Por parte de GeCode se evaluarán las clases estándar, ya que son las mismas aplicables a GeCode/J. y solo se harán referencia algunas de las clases, ya que se dispone de una gran cantidad para acciones muy específicas [62].

5.3.1 Justificación del uso de Modelos.

Entendemos como Modelo, a todo aquel problema que pueda ser modelado e implementado utilizando programación con restricciones y luego permita realizar algún tipo de comparación. Los problemas seleccionados, pueden pertenecer a diferentes ámbitos, en muchos casos son problemas lógicos o matemáticos, de aplicaciones en investigación operativa (generación de horarios, scheduling, etc.), en bioinformática (identificación y ordenación de secuencias genéticas), en ingeniería electrónica (diseño de circuitos), telecomunicaciones (asignación de frecuencias), en informática (bases de datos y sistemas de recuperación de la información, lenguaje natural,

planificación, etc.), en problemas de diseño y configuración, empaquetamiento, etc. Estos problemas se utilizan para comprobar las características de un algoritmo o de un determinado lenguaje de programación. Algunos de estos se pueden observar en el CSPLib [30] [70].

En el CSPLib se encuentran una serie de problemas numerados y clasificados según área temática. Entre estas podemos encontrar (se hace referencia a sus nombres en ingles);

Programación: Entre estos problemas se encuentran: Secuenciación de coches, Mystery Shopper. Social golfer problem. ACC basketball schedule, progressive party problem, traffic lights, darts tournament, bus driver scheduling., round robin tournaments, balanced academic curriculum problem. rehearsal problem. a distribution problem with Wagner-Whitin costs. meeting scheduling problem. supply chain coordination [70].

Diseño, configuración y diagnóstico: template design. vessel loading, perfect square placement, water bucket problem, magic hexagon, balanced incomplete block designs, rack configuration problem, warehouse location problem, fixed length error correcting codes, steel mill slab design problem, circuit fault diagnosis, differential fault diagnosis, covering arrays y minimum energy broadcast [70].

Empaquetado y particionamiento: template design, vessel loading, perfect square placement, social golfer problem, Schur's lemma, Ramsey numbers, water bucket problem, steel mill slab design problem, a distribution problem with Wagner-Whitin costs.

Asignación de frecuencias: low autocorrelation binary sequences, Golomb rulers, all-interval series.

Combinatoria matemática: Low autocorrelation binary sequences, Golomb rulers, all-interval series, perfect square placement, Schur's lemma, Ramsey numbers, magic squares and sequences, magic hexagon, Langford's number problem, Lam's problem, balanced incomplete block designs, prime queen attacking problem, word design for DNA computing on surfaces, Molnar's problem, fixed length error correcting codes, n-fractions puzzle, Steiner triple systems, covering arrays [70].

Juegos y rompecabezas: social golfer problem, nonagrams, Solitaire Battleships, water bucket problem, magic squares and sequences, darts tournament, crossfigures, magic hexagon, Langford's number problem, alien tiles, prime queen attacking problem, maximum density still life, peg solitaire, n-fractions puzzle [70].

Bioinformática: word design for DNA computing on surfaces.

La problemática principal esta en justificar el uso de un determinado Modelo a la hora de realizar una prueba de un algoritmo o lenguaje. La solución a esta interrogante debiese estar directamente relacionada con las características

que queremos resaltar de ese determinado algoritmo o lenguaje. Por lo tanto en este informe realizaremos una primera clasificación de los diferentes problemas, según se quiera evaluar la eficiencia o la escalabilidad. A continuación se presenta un pool de algunos problemas clásicos clasificados según estos criterios [70].

Modelos para medir la eficiencia: Para resolver un problema pueden existir varios algoritmos e implementados en diferentes lenguajes. Por tanto, se busca elegir el mejor. Si el problema es sencillo o no hay que resolver muchos casos se podría elegir el más fácil. Si el problema es complejo o existen muchos casos habría que elegir el algoritmo que menos recursos utilice. El recurso más importante es el tiempo de ejecución. Al hablar de la eficiencia nos referiremos a lo rápido que se ejecuta, la eficiencia de un algoritmo dependerá, en general, del "tamaño" de los datos de entrada. Algunos problemas que nos permiten evaluar la eficiencia son [80]:

Send more Money: El típico problema criptográfico "send+more = money" consiste en asignar a cada letra {s, e, n, d, m, o, r, y} un dígito diferente del conjunto {0, …, 9} de forma que se satisfaga: send + more = money.

$$\begin{array}{r} S\ E\ N\ D \\ \underline{+\ M\ O\ R\ E} \\ M\ O\ N\ E\ Y \end{array}$$

La manera más fácil de modelar este problema es asignando una variable a cada una de las letras, todas ellas con un dominio {0,…, 9} y con las restricciones que obliguen a que todas las variables tomen valores distintos y con la correspondiente restricción para que se satisfaga "send + more = money".

Ecuación (n): Resolución de sistemas de ecuaciones lineales, por ejemplo Ecuación (10), se resuelve un sistemas de 10 ecuaciones lineales con siete variables entre {0…, 10}

SRQs: Los puzzles SRQ (Self Referencial Quizzes) fueron inventados por Jim Propp alrededor de 1993. El cuestionario SRQ original contiene 20 preguntas con 5 alternativas cada una, casi todas ellas se referencian a si mismas. El problema consiste en encontrar la solución correcta para cada pregunta. Las restricciones en SRQ son tan fuertes que conducen a una única solución casi determinísticamente.

Modelos para medir la escalabilidad: La escalabilidad es la propiedad deseable de un sistema, que indica su habilidad para, o bien manejar el crecimiento continuo de trabajo de manera fluida, o bien para estar preparado para hacerse más grande sin perder calidad de sus resultados. En general, es la capacidad de cambiar su

tamaño o configuración para adaptarse a las circunstancias cambiantes. Los Modelos que se presentan a continuación, tienen la capacidad de poder ser modificados para poder testear la escalabilidad de los lenguajes [80].

N-reinas: Este problema consiste en colocar N reinas en un tablero de ajedrez de dimensión NxN, de forma que ninguna reina esté amenazada. De esta forma no puede haber dos reinas en la misma fila, misma columna, o misma diagonal. Si asociamos cada columna a una variable y su valor representa la fila donde se coloca una reina. Este problema aumenta considerablemente mientras se aumente el valor de N.

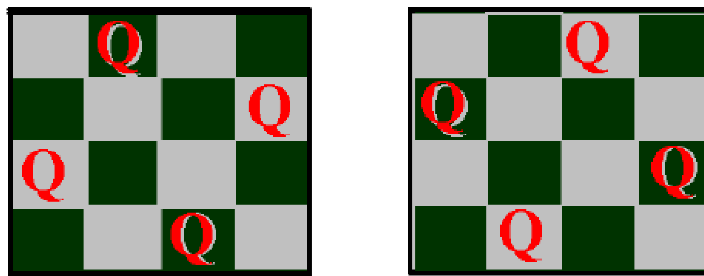

Figura 5.2: Solución para 4 reinas

Cuadrados Mágicos: Un cuadrado mágico es la disposición de una serie de números enteros en un cuadrado o matriz de tal forma que la suma de los números por columnas, filas y diagonales sea la misma, la constante mágica. Usualmente los números empleados para rellenar las casillas son consecutivos, de 1 a n^2, siendo n el número de columnas y filas del cuadrado mágico (en el ejemplo, n = 3). Donde el numero mágico es $(n*(n^2+1))/2$.

4	9	2
3	5	7
8	1	6

Figura 5.3: Cuadrado Mágico n=3, número mágico 15.

Coloreo de mapas: Este problema parte de un conjunto de colores posibles para colorear cada región del mapa, de manera que regiones adyacentes tengan distintos colores. En la formulación del CSP, definimos una variable por cada región del mapa, y el dominio de cada variable es el conjunto de colores disponible. Para cada par de regiones contiguas existe una restricción sobre las variables correspondientes que no permite la asignación de idénticos valores a las variables. En el caso de la Figura 5.4, tenemos un mapa con cuatro regiones x,y,z,w para ser coloreadas con los posibles colores Rojo, Verde, Azul.

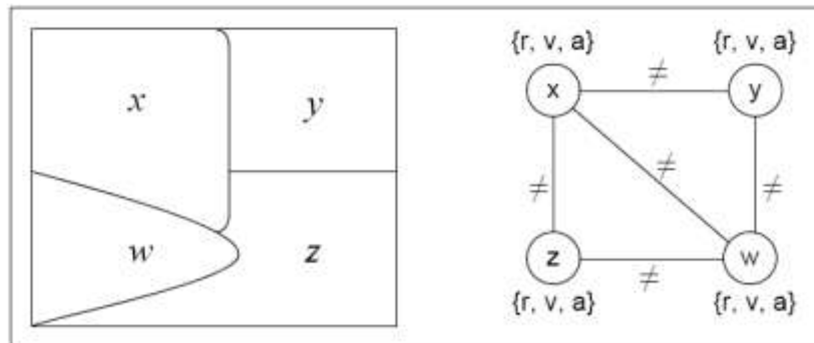

Figura 5.4: Modelo del problema de coloreo de mapa

Sudoku: El sudoku es un claro ejemplo de representación del puzzle como un problema de satisfacción de restricciones. Este problema se publicó en Nueva York en el año 1979 bajo el nombre de "Number Place" y se hizo popular en Japón con el nombre de sudoku ("*Su*dji wa *doku*shin ni kagiru": "los números deben ser sencillos" o "los números deben aparecer una vez"). En el problema del sudoku como se muestra en la figura 5.5 hay que rellenar las casillas de un tablero de 9 x 9 con números del 1 al 9, de forma que no se repita ningún número en la misma fila, columna, o subcuadro de 3 x 3 que componen el sudoku.

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

Figura 5.5: Ejemplo de Sudoku 9x9

Según hemos visto la clasificación anterior es aplicable a cualquier tipo de algoritmo o lenguaje de programación, pero no son de gran utilidad, para comprobar las capacidades de un lenguaje determinado de programación con restricciones, en este caso deberíamos buscar aquellos problemas que nos permitan comprobar las características deseables de un determinado lenguaje. Como se dijo en la sección anterior, los lenguajes a comparar serán evaluados según su capacidad de poder definir restricciones, definir propagadores, asignar valores a las variables, las operaciones sobre el dominio. La eficiencia ya fue justificada en la sección anterior.

Modelos para evaluar la definición de restricciones: Como ya hemos visto la definición de restricciones nos va a permitir una mayor expresividad en la representación del problema, permitiendo así definir las restricciones unarias, restricciones binaria, restricción ternaria y n- arias. Las características deseables de estos problemas son aquellos que nos permitan tener restricciones de variados tipos, tanto unarias, ternarias como n-arias. Entre estos problemas podemos encontrar: send more Money, Ecuación (n), n-Reinas, Coloreo de mapas, Sudoku, de los ya explicados [80]. Además se pueden nombrar otros problemas del CSPLib como:

Meeting scheduling problem: El problema de programación de reuniones (MSP) se describe a continuación como un problema de satisfacción de restricciones (CSP). Sin embargo, una de sus características más interesantes es el hecho de que se trata de un CSP distribuido. Informalmente, un conjunto de los agentes quieres realizar una reunión. Ellos búsqueda el horario de una posible reunión que cubra con cada una de sus restricciones privadas, las cuales necesitan ser satisfechas y deben cumplir con la llegada a la reunión a tiempo. (Entre las distintas reuniones del mismo agente) [81].

La definición general de la familia MSP es el siguiente:

- Un conjunto S de m agentes.
- Un conjunto de T de n reuniones.
- La duración de cada reunión m_{ii}.
- Cada reunión m_i es asociada a un conjunto s_i de agentes en S que asisten a dicha reunión.
- Consecuentemente, cada agente tiene un conjunto de reuniones a las que tienen que asistir.
- Cada reunión es asociada con una locación.
- La programación de un time-slots para las reuniones en T debe permitir que los agentes participantes, puedan viajar entre sus reuniones para asistir.

Balanced Academia curriculum problem: El BACP consiste en diseñar un currículo académico equilibrado mediante la asignación de los períodos a los cursos de manera que la carga académica de cada período es equilibrado, es decir, lo más similar posible. El currículo debe obedecer los siguientes reglamentos administrativos y académicos [82].

- Currículo académico: un currículo académico se define por un conjunto de cursos y una serie de requisito previo relaciones entre ellos.
- Número de períodos: los cursos deben ser asignados dentro de un número máximo de períodos académicos.
- Carga Académica: cada curso tiene asociado un número de créditos o unidades que representan en el mundo académico el esfuerzo necesario para terminar con éxito el curso.
- Requisitos previos: algunos cursos pueden tener otros cursos como requisitos previos.
- Carga académica mínima: una cantidad mínima de créditos académicos por cada período que tiene la obligación de tomar un estudiante a tiempo completo.
- Máxima carga académica: un importe máximo de créditos académicos por cada período se permite, a fin de evitar la sobrecarga.
- Número mínimo de cursos: un número mínimo de cursos por cada período tiene la obligación de tomar un estudiante a tiempo completo.
  Número máximo de los cursos: un número máximo de cursos por cada período se permite, a fin de evitar la sobrecarga.

Modelos para evaluar propagadores: El objetivo de la Propagación de Restricciones es restringir el conjunto de posibles valores que pueden tomar las variables, mediante la aplicación de las restricciones, hasta que finalmente se encuentre un solo valor para cada variable. El conjunto de posibles valores se almacena en una estructura denominada depósito de restricciones (constraint store). De acuerdo a esto lo que se busca es utilizar aquellos problemas que tengan dominios amplios para cada variable, así poder comprobar el poder de los propagadores y de las distintas heurísticas que apliquen cada propagador, por ejemplo un problema con dominios binarios, no serian tan buena forma de medir los propagadores, en cambio un problema con dominios amplios para las variables puede ser una buena prueba para la medición. Entre estos problemas podemos encontrar: Ecuación(n), n-Reinas, Sudoku, Coloreo de mapas, BACP, Cuadrados mágicos. Entre los ya explicados anteriormente. Se puede nombrar otro problema del CSPLib como:

Solitario: Se juega en un tablero con un número de agujeros. En la versión en Inglés del juego, el tablero está en la forma de una cruz con 33 hoyos:

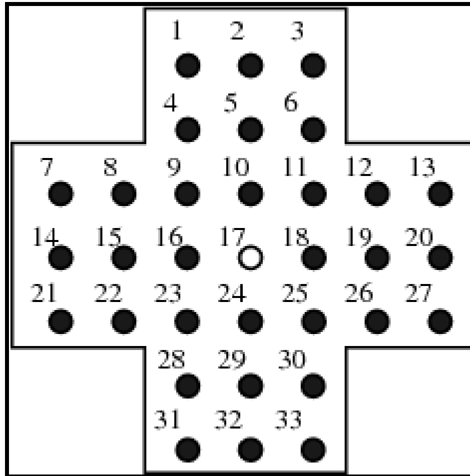

Figura 5.6: Tablero de solitario de 33 hoyos.

Las Clavijas (X) están dispuestos en el tablero de manera que, al menos, se pueda mover una clavija saltando sobre otra clavija en forma horizontal o vertical a una posición vacía. Eliminando la clavija saltada. El objetivo del juego es dejar solamente una clavija en el centro del tablero.

Modelos para evaluar la asignación de valores a las variables: La asignación de valores a las variables, es la asignación de un determinado valor a una variable de todos los valores posibles del dominio, esta es una característica medible con cualquier tipo de problema, ya que independiente del dominio siempre se necesita realizar la asignación de valores a una variable, ahora esta operación va de la mano la aplicación de enumeración aplicado para resolver dicho problema. Entre los problemas que permiten comprobar esta característica encontramos: Send more Money, Sudoku, n-Reinas [80]. Y otros como:

Problema de los cubos de agua: Se le da un cubo con 8 unidades de agua, y dos cubos vacíos que pueden contener 5 y 3 unidades, respectivamente. Usted está obligado a dividir el agua en dos vertiendo cubos de agua entre (es decir, para terminar con 4 unidades en el cubo de 8 unidades, y 4 unidades en el de 5 unidades cubo).
¿Cuál es el número mínimo de las transferencias de agua entre los cubos?

Modelos para evaluar las operaciones sobre el dominio: Una operación de dominio apoya simultáneos accesos o actualizaciones a múltiples valores de un dominio de una variable. En muchos lenguajes esto es proporcionado por el apoyo a de un conjunto de tipos de datos abstractos para los dominios de las variables. Los problemas deseables para medir las operaciones sobre el dominio son aquellas que requieran un alto grado de acciones sobre el dominio, ya que permitirán evaluar la capacidad del lenguaje para definir acciones sobre estos

valores. Entre estos problemas podemos encontrar: Ecuación (n), n-Reinas, Sudoku, Coloreo de mapas, Solitario. Ya explicados anteriormente.

5.3.2 Implementación de Modelos.

Para realizar las pruebas sobre los lenguajes de programación Eclipse y GeCode se han seleccionado un par de problemas o modelos, los cuales han sido seleccionados en base a las características de que se necesitan explotar de cada lenguaje de programación.

En esta selección para limitar la cantidad de problemas a modelar se va a utilizar el problema de N-reinas, ya que este problema sirve para explotar la mayoría de las características que se desean visualizar de cada lenguaje de programación, además no tiene una implementación compleja, ya que no se desea incurrir en modelos difíciles de implementar que sumen una dificultad inherente la cual no es objetivo de estudio. Otro problema que se desea implementar es el problema de cuadrados mágicos, ya que al igual que el problema de N-reinas es más simple de modelar y sirve para explotar una gran gama de indicadores. Por otro lado Se cuenta con definiciones e implementaciones básicas de estos problemas para cada lenguaje de programación estudiado.

5.3.2.1 Modelo de N-reinas:

Para resolver el problema, se realiza el modelo de la solución teniendo en consideración lo siguiente:

- La Función objetivo no tiene un uso práctico ya que se sabe a priori el valor de n. Así, el problema se presenta como un CSP.
- Dos reinas no pueden estar en la misma fila
- Dos reinas no pueden estar en la misma columna
- Dos reinas no pueden estar en la misma diagonal.

Dado lo anterior, se propone el siguiente modelo usando variables binarias:

Variables: $X_{ij} = 1$ *tiene un valor 1 si hay una reina en la posición (i,j)*
Dominios: {0,1}
Restricciones:

$$\sum_{J=1}^{n} x_{ij} = 1 \qquad ;\forall i = \{1,\ldots,n\}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad ;\forall j = \{1,\ldots,n\}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} \leq 1 \qquad ; i + j = k, \forall k = \{3,\ldots,2n\text{-}1\}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} \leq 1 \qquad ; i - j = k, \forall k = \{1\text{-}n,\ldots,n\text{-}1\}$$

Hay otra posible representación del problema en la cual no es necesario definirlo con variables booleanas, sino que asumiendo *n* variables y cada una puede tener un valor entre 1 y *n*. como se observa a continuación:

Variables: $\{X_i\}$, i = 1...N

Dominio: {1, 2, 3,…, N}

Restricciones:

$$\forall x_i, x_j, i \neq j :$$
$$x_i \neq x_j$$
$$x_i - x_j \neq i - j$$
$$x_j - x_i \neq i - j$$

5.3.2.2 Modelo de Cuadrado Mágico:

Para resolver el problema de los cuadrados mágicos debemos considerar que hay una constante mágica, la cual esta dada por el numero de *n* del cuadrado mágico NxN, este se conoce como numero mágico y es igual a $(n*(n^2+1))/2$.

- Se tiene un conjunto de n variables que pueden asumir un valor entre 1 a N.
- Todas las variables deben asumir valores distintos.
- La suma de las n filas debe ser igual al numero mágico.
- La suma de las n columnas debe ser igual al número mágico.
- La suma de sus diagonales debe ser igual al número mágico.

Variables: $\{X_{ij}\}$, $i = 1,\ldots,$ N y $j = 1,\ldots,$ N

Dominio: {1, 2, 3,…, NxN}

Restricciones:

$$\forall x_{ij}, x_{km}:$$

$$x_{i,j} \neq x_{km} i \neq k \ y \ j \neq m$$

$$\sum_{\mathrm{j=1}}^{\mathrm{n}} x_{i,j} = MN \quad \mathrm{i = (1,..,N),\ donde\ MN\ es\ número\ mágico}$$

$$\sum_{\mathrm{i=1}}^{\mathrm{n}} x_{i,j} = MN \quad \mathrm{j = (1,..,N),\ donde\ MN\ es\ número\ mágico}$$

$$\sum_{\mathrm{i=1}}^{\mathrm{n}} x_{i,i} = MN$$

$$\sum_{\mathrm{i=1}}^{\mathrm{n}} x_{i,n+1-i} = MN$$

5.3.3 Comparación práctica de los lenguajes usando Modelos.

Los modelos antes descritos para los problemas seleccionados, en este caso el N-reinas y el problema de Cuadrados Mágicos, se han implementado usando los dos lenguajes de programación con restricciones en estudio.

Las pruebas se realizan principalmente para comprobar la eficiencia de los lenguajes de programación, así como la expresividad en la representación de la solución para cada lenguaje. Para ello es necesario comparar inicialmente los lenguajes en cuanto a las funciones utilizadas para la representación de la solución, así como la claridad con la que se puede llevar el problema desde su definición para CSP a la implementación propiamente tal usando los criterios que fueron analizados previamente. Otro punto interesante de analizar son las líneas de código necesarias para implementar la solución, pues puede resultar relevante a la hora de realizar una implementación en la cual el tiempo de desarrollo es critico, esta característica será emparejada con la capacidad de reutilización de código para cada lenguaje. Además de los puntos antes descritos para comparar la eficiencia se realizará la comparación en función del tiempo de ejecución, el uso de memoria, uso de disco y número de backtrack para encontrar la primera solución. Por ultimo se buscará representar las diferencias en ambos lenguajes haciendo uso de la medición de número de backtrack para cada problema en cada lenguaje. En función de la evaluación obtenida tras la medición de los criterios estaremos en condiciones de realizar la evaluación de los criterios sujetos a comprobación experimental haciendo uso de la tabla de ponderación y evaluación propuesta en secciones anteriores.

Hardware y compiladores

Las experimentaciones se realizaron sobre una maquina con un procesador Intel Core Duo de 2 Ghz y 2 GB de memoria RAM, bajo un sistema operativo Windows XP. Los compiladores utilizados, para el caso de Eclipse fue la versión 5.10 y se programó haciendo uso de TkEclipse. Mientras que para el trabajo con GeCode se utilizo el compilador GeCode-2.2.0-Qt-x86 con GeCode\J, Utilizando para java la versión de JDK 1.6.0_02, todas versiones para Windows.

Líneas de Código y reutilización

A continuación se presentan los resultados obtenidos al comparar las líneas de código entre la implementación de Eclipse y GeCode, esta comparación corresponde a la comprobación de la reutilización de código. En el caso de Eclipse se tiene que para el N-Reinas son suficientes alrededor de 32 líneas de código, aunque este numero corresponde al código más el código agregado para entregar los resultados estadísticos (código adicional para generara los output necesarios), El tener menos líneas de código que la implementación en GeCode/J no necesariamente dice que sea más representativo o reutilice más código que GeCode\J. Ya que la reutilización de código en Eclipse es limitada, solo se pueden utilizar las funciones pre definidas en las librerías y las propias solo se pueden definir dentro del mismo archivo o utilizando *use_module*. Por otro lado en GeCode\j si bien los programas pueden ser escritos utilizando más líneas de código, pero eso es inherente a la Programación Orientada a Objeto, el beneficio esta dado por la facilidad en la reutilización de código, el cual puede ser propio o de las librerías que posee GeCode\j. De echo es esta implementación se llamo a una clase *Option.class*, la cual sirve para mostrar los resultados estadísticos producto de la ejecución. Las líneas de código solo fueron medidas para el archivo fuente principal y no aquellos archivos de importados en cada instancia.

Problema	Eclipse	GeCode\J
N-Reinas	32	72
Cuadrado Mágicos	45	81

Tabla 5.3: Líneas de código utilizadas en cada lenguaje.

Uso de Disco

Se hizo una comparación en función a la cantidad de disco duro que fue utilizado por cada lenguaje y cada instancia. Cabe mencionar que en la implementación de GeCode/J en el problema de N-reinas y Cuadrados Mágicos se hizo uso de un archivo adicional que es la clase Option.class en ese caso el archivo utilizó alrededor de 11 KB de memoria en disco. Para ambos problemas el archivo utilizado ocupo alrededor de 2 KB de disco más un archivo .class que peso alrededor de 3KB. En el caso de Eclipse solo se utilizó un archivo por instancia y estos usaron alrededor de 1,5 KB de disco. Esto se representa en la Tabla 5.4. Por otra parte también se informa el espacio utilizado por cada compilador en el disco duro. Para el caso de Eclipse ocupa en disco alrededor de 79,5 MB y GeCode/J utiliza en disco alrededor de 68,8 MB. Hay que mencionar que además para ejecutar GeCode/J se utiliza un compilador externo de Java.

Problema	Eclipse	GeCode\J
N-Reinas	1,1KB	1,5 KB (2,8 KB)
Cuadrado Mágicos	1,5KB	1.86 KB (3,1KB)

Tabla 5.4: Cantidad de disco utilizado en cada implementación

Uso de Memoria

En cuanto al uso de memoria de cada instancia en GeCode indicamos cual fue el peak de memoria utilizado en cada ejecución representada en kilobytes. Para Eclipse la ejecución no fue exitosa en un tiempo de 3 minutos para las instancias más grandes de cada modelo. Por eso no se obtuvo un resultado. Hay que mencionar que esto se debe al utilizar las estrategias de selección de variable y valor básicas, ya que realizando otras heurísticas es los solver son capaces de obtener mejores soluciones que las obtenidas en este estudio.

Problema	Eclipse	GeCode\J
N-Reinas (10/15/20/80)	0,7/1/1,3/---	18/18/22/165
Cuadrado Mágicos (3/4/6)	2,5/3,6/--	14/35/100

Tabla 5.5: Memoria utilizada en la ejecución de cada instancia expresada en KB.

Tiempo de Ejecución

Eclipse: La reutilización en el lenguaje Eclipse, no es muy efectiva, ya que las funciones tienen que ser declaradas en el mismo archivo, y el dominio de una variable es la regla en la cual esta definida de otra forma se puede crear un modulo. A pesar de eso se pueden utilizar una gran cantidad de funciones pre definidas en las librerías. Para el caso de N-reinas el mas importante es el uso de *ic*, por ejemplo:- lib(*ic*). Otra ventaja del uso de Eclipse es que las implementaciones pueden para instancias pequeñas utilizar pocas líneas de código. Pero tiene el inconveniente que las instancias de un problema deben ser implementadas en el mismo código, lo cual puede tomar una extensión en las líneas de código utilizadas.

GeCode: En el caso de GeCode gracias a las ventajas de la orientación a objeto permiten reutilizar mejor clases ya definidas, así como definir nuevas clases específicas a partir de las ya existentes. En este caso se importan las clases org.gecode.*, org.gecode.Gecode.* y org.gecode.GecodeEnumConstants.*.

Para estandarizar el proceso de propagación y enumeración se utilizan enumeraciones estándar definidas en cada lenguaje en este caso se muestran los resultados implementando la soluciones con *naive* para la enumeración. La implementación de las soluciones corresponden a algoritmos básicos que no implementar heurísticas de optimización del proceso de búsqueda, solo se busca satisfacer las restricciones. Para las pruebas se utilizan las configuraciones estándar de cada solver. A continuación se muestra el tiempo de ejecución en segundos de las instancias utilizadas.

Instancias	Eclipse	GeCode
10 Reinas	0,02	0,17 /5
15 Reinas	0,046	0,4 /12
20 Reinas	3.38	0,31/9
80 Reinas	sobre	2,8 /81

Tabla 5.6: Tiempo de ejecución para N-reinas.

Instancias	Eclipse	GeCode
C.Magico 3	0,008	0,1/3
C.Magico 4	0,015	2,0/58
C.Magico 6	sobre	0,86/25

Tabla 5.7: Tiempo de ejecución para Cuadrado Mágico.

Como puede observarse en las Tablas Eclipse tiene soluciones en un tiempo menor para las instancias pequeñas, pero a medida que aumenta la complejidad del problema, cuesta más tiempo el encontrar la primera solución. En el caso de GeCode toma más tiempo para instancias pequeñas, pero su eficiencia es constante a medida que aumenta la complejidad las soluciones y aun así sigue obteniendo soluciones factibles en un tiempo con aumento mas lineal que en el caso de Eclipse. Estas variaciones y la incapacidad de Eclipse de encontrar soluciones para algunas instancias pueden deberse a que no se han incorporado estrategias para selección de valor y de variables mas acordes a los problemas. No se han hechos estas modificaciones ya que el objetivo de este estudio no es obtener el mejor resultado y compararlo con otros benchmark sino simplemente analizar la implementación del modelo y ver la capacidad de resolución de los lenguajes utilizados, para satisfacer los criterios de comparación propuestos.

Número de Backtracks

Como una forma de poder comparar la capacidad de ambos lenguaje de encontrar soluciones se presenta el número de backtracks realizados en ambos lenguajes para distintas instancias del N-Reinas.

Instancias	Eclipse	GeCode
10 Reinas	6	9
15 Reinas	73	69
20 Reinas	10026	36
80 Reinas	sobre	21

Tabla 5.8: Número de Backtracks para encontrar la primera solución en N-reinas.

En este caso podemos observar que para la instancia de 10 Reinas se encuentra una solución factible tan solo con realizar 6 backtracks, mientras que para el caso de GeCode es necesario realizar 9, esto también puede explicar en parte la diferencia en el tiempo de ejecución de cada instancia, en la cual GeCode tomas más tiempo que Eclipse. Por otro lado para la instancia 20 Reinas Eclipse toma alrededor de 10026 backtracks mientras que GeCode toma 36 backtracks, pero el tiempo de ejecución de GeCode es menor que el tiempo de Eclipse, lo cual reafirma lo dicho en el punto anterior que las soluciones en GeCode, tienen un tiempo de ejecución más constante que Eclipse. Eclipse para la parametrización con que se hicieron las pruebas funciona muy bien para instancias pequeñas pero cuando aumenta la complejidad se le dificulta poder encontrar una solución, lo cual se muestra como *sobre,* para indicar que supera un tiempo de 10 veces mayor al máximo tiempo de GeCode para encontrar una solución .

5.4 Evaluación y Ponderación final del Caso: Eclipse y GeCode/J.

En esta sección haremos un análisis de los resultados obtenidos en las comparaciones experimentales y las teóricas o sujetas a bibliografía, con los datos obtenidos realizaremos la ponderación de los criterios en función de la comparación de los criterios más prometedores para la comparación entre lenguajes y puntuaremos cada criterio de acuerdo a los estudios realizados en las secciones previas. Posteriormente compararemos los puntajes en función a la tabla propuesta en la sección 4.

A continuación se procederá a asignar una puntuación por cada grupo criterios a Eclipse y GeCode para ver el funcionamiento de la plantilla propuesta de comparación. Además se debe fijar las ponderaciones de cada criterio en función de la relevancia que estos tienen para el objetivo de la comparación. El enfoque principal que se desea dar a esta comparación es potenciar todos aquellos criterios que evalúan las características deseables para en lenguaje de programación con restricciones. Es por esto que se ponderará mayormente al grupo de criterios de eficiencia, manejo de restricciones y expresividad en ese orden de relevancia. La ponderación de estos criterios de acuerdo a la importancia que nosotros hemos asignado es la siguiente:

Expresividad	0,2
Eficiencia	0,4
Desarrollo de restricciones	0,3
CRITERIOS EXTERNOS	0,1
TOTAL	1

Tabla 5.9: Ponderación de cada grupo de criterios para el Caso: Eclipse y GeCode/J

Ahora procederemos a puntuar cada criterio de acuerdo a la comparación realizada en las secciones previas estos puntajes han sido calculados de acuerdo al sistema de puntuación de criterios establecido en la sección 4. Y la ponderación de cada criterio es en base la estimación de la importancia relativa de cada criterio para esta evaluación en particular, pero usando como guía la propuesta de tabla de ponderación presentada en la tabla 4.4. Para simplificar la comparación se presentará la una tabla con los valores y ponderaciones de cada criterio.

	Eclipse			GeCode		
CRITERIOS	Puntaje	Pond.	Total	Puntaje	Pond.	Total
CRITERIOS INTERNOS						
Expresividad						
Líneas de código	69,2	0,05	3,46	30,7	0,05	1,535
Tipificación	80	0,05	4	80	0,05	4
Tipos de datos y estructuras	60	0,2	12	80	0,2	16
Def de dominios	100	0,15	15	100	0,15	15
Def de restricciones	50	0,15	7,5	100	0,15	15
Operaciones sobre dominio	100	0,15	15	100	0,15	15
Integración	50	0,2	10	70	0,2	14
Sencillez	60	0,05	3	70	0,05	3,5
SUB TOTAL		1	69,96		1	84,035
Eficiencia						
Tiempo de ejecución	46,8	0,3	14,04	53	0,3	15,9
Uso de memoria	95,6	0,1	9,56	4,3	0,1	0,43
Uso de disco	57	0,1	5,7	43	0,1	4,3
Nº Backtrack	36,46	0,3	10,938	63,54	0,3	19,062
Reutilización	30	0,1	3	80	0,1	8
Seguridad de tipos	100	0	0	100	0	0
Tipo de traducción	100	0	0	100	0	0
Portabilidad	30	0	0	80	0	0
Recolección de basura	100	0	0	100	0	0
Comprobación de tipos	100	0	0	100	0	0
A prueba de fallos	100	0,1	10	100	0,1	10
SUB TOTAL		1	53,238		1	57,692
Desarrollo de restricciones						
Definición de propagadores	70	0,2	14	80	0,2	16
Definición de nuevos propagadores	50	0,2	10	30	0,2	6
Heurísticas de selección de valor	70	0,1	7	80	0,1	8
Generar nuevas heurísticas de selección de valor	90	0,2	18	60	0,2	12
Heurísticas de selección de variable	70	0,1	7	80	0,1	8
Generar nuevas heurísticas de selección de variable	90	0,2	18	60	0,2	12
SUB TOTAL		1	74		1	62
CRITERIOS EXTERNOS						
Paradigma de programación	100	0,3	30	100	0,3	30
Tipo de licencia	100	0	0	100	0	0
Integración a alto nivel de abstracción	20	0,3	6	40	0,3	12
Estándares	100	0	0	100	0	0
Grado de penetración en el mercado	50	0,1	5	60	0,1	6
Finalidad de uso	80	0,3	24	70	0,3	21
Soporte	90	0	0	70	0	0
SUB TOTAL		1	65		1	69

Tabla 5.10: Puntuación y ponderación de cada criterio en el Caso: Eclipse y GeCode/J.

Como podemos observar por cada criterio se asigno un puntaje, el cual se calculo según la comparación de los criterios, este puntaje fue multiplicado por una ponderación. Esta ponderación puede variar según la relevancia que se le quiera dar al criterio. Una ves sumados los puntajes de cada conjunto de criterios podemos comparar los lenguajes en función de este subtotal para el grupo que se quiera analizar. Algunos criterios tienen ponderación 0 esto es porque no se consideran importantes para la evaluación.

	Eclipse			**Gecode**		
Grupos	Sub totales	Pond	Total	Sub totales	Pond	Total
Expresividad	69,96	0,2	13,99	84,04	0,2	16,81
Eficiencia	53,24	0,4	21,3	57,69	0,4	23,08
Desarrollo de restricciones	74	0,3	22,2	62	0,3	18,6
CRITERIOS EXTERNOS	65	0,1	6,5	69	0,1	6,9
TOTAL EVALUACION		1	63,99		1	65,38

Tabla 5.11: Puntuación y ponderación de grupos de criterios.

Finalmente podemos realizar la comparación entre cada grupo de criterios en este caso particular de acuerdo a la ponderación que se realizó podemos observar que GeCode obtiene una mayor puntuación en expresividad y en eficiencia, esta diferencia se reduce cuando se realiza la comparación en función de los criterios externos a pesar que de todas formas GeCode tiene mejor puntuación que Eclipse. En el único punto de comparación donde Eclipse supera a GeCode es en Manejo de restricciones, esto puede deberse a que Eclipse es más maleable para crear y modificar las heurísticas de selección de valor y de variable. Luego se calcula la evaluación final para cada lenguaje según la ponderación que se le asigne a cada grupo de criterios. Con lo cual GeCode resulta mejor evaluado que Eclipse con un total de 65,38 sobre un 63,99 de Eclipse. Tal como se observa en la tabla 5.11.

Podríamos realizar una nueva ponderación de los grupos de criterios (o de cada criterio si así se necesitara), con lo cual podrían variar los resultados de esta evaluación (conservando el puntaje asignado a cada criterio), esto es parte de la capacidad de parametrización que tiene esta plantilla de comparación, la cual se puede modificar de acuerdo a las necesidades de cada evaluación en función de los criterios que le parezcan mas relevantes para el uso final que se desee dar al lenguaje.

Conclusiones

Conclusiones.

Se ha logrado concretar de manera satisfactoria los objetivos planteados al inicio de este proyecto. La realización de un estudio para crear un marco general para realizar comparaciones entre distintos lenguajes de programación con restricciones.

Para esto se ha presentado de manera formal el estado de arte de la programación con restricciones principal tema de estudio para el cual se gestó este trabajo. Junto con esto se establecieron las características de los diferentes paradigmas de programación utilizados para trabajar con Programación con Restricciones. De esta forma nos enlazamos con la Programación Lógica y Programación Orientada a objetos, dando lugar a otro tema relevante que es la Programación Lógica y Orientada a Objetos con restricciones. El enlace con la Programación Lógica con Restricciones es necesario dado que los lenguajes de programación que se analizan, perteneces a Lenguajes Declarativos Lógicos y son los más utilizados para la Programación con Restricciones gracias a las características propias de sus solvers. Además se toma la contraparte con Los Lenguajes Orientados a Objetos, que se caracterizan por incorporar una serie de librerías en C++ o en Java, que permiten implementar la Programación con Restricciones.

Los Lenguajes de Programación que se tienen son tan variados, que el decidir sobre que lenguaje programar se vuelve una tarea dificultosa, agregando una complejidad mayor al problema que se desea resolver usando Programación con Restricciones. Es por esto que se planteó una serie de criterios de comparación, los cuales representan características deseables de los Lenguajes para Programación con Restricciones. Además se genero un sistema de ponderación de criterios para premiar aquellos criterios que parezcan más relevantes que otros según los objetivos del evaluador. Lo que entendemos como un sistema de comparación parametrizable.

Se logró de forma exitosa probar este sistema de comparación usando un Caso para comparar dos lenguajes de programación, que perteneces a paradigmas de programación diferentes como son Eclipse por parte de los Lógicos, y por parte de Programación Orientada a Objetos GeCode. Estos lenguajes fueron comparados y evaluados en función de cada criterio. Para luego obtener una puntuación final de evaluación de los lenguajes, resultando al termino de este proceso con mayor puntuación GeCode, esto si de acuerdo a la ponderación establecida para este caso particular de evaluación. Esta evaluación podría tener diferencias si se establecen otras ponderaciones para los criterios. Estas modificaciones son las que le dan un carácter parametrizable a este sistema de comparación.

Se pudo observar que Eclipse proporciona una serie de librerías que permiten tener una mayor representabilidad de los problemas y sus restricciones, así como la creación de propagadores y heurísticas. GeCode proporciona un número aun mayor de clases, las cuales permiten tener un gran manejo en las restricciones. Clases definidas específicamente para la manipulación de dominios y una gran cantidad de propagadores definidos. Se establecen ciertos puntos importantes dentro de la comparación de estos dos lenguajes. Se observó que es más fácil representar un modelo CSP en Eclipse, pero para la resolución del problema GeCode cuenta con una mayor cantidad de métodos para definir propagadores. Además GeCode cuenta con una mayor solides para resolver problemas, dado el tiempo que demora en encontrar la primera solución en comparación Eclipse.

Podemos referirnos además a la hipótesis planteada al inicio de este trabajo, donde se estipulaba que los Lenguajes Orientados a Objetos proporcionan mejorías en el modelado y en la implementación de CSP respecto a los Lenguajes Lógicos. En función a las comparaciones realizadas en este estudio, se puede establecer que los Lenguajes Lógicos tienen una mayor expresividad que los Orientados a Objetos, lo cual facilita mayormente el proceso de modelado, O sea traducir el modelo del problema al lenguaje de programación, pero en términos de las funciones que se pueden utilizar para obtener una implementación eficiente, los Lenguajes Orientados a Objetos son más estables y se pueden obtener mejores resultados para modelos con la misma parametrización. Ahora esto podría variar haciendo uso de distintas estrategias de búsqueda de soluciones que se adapten mejor a los modelos resueltos.

Se realiza un breve estudio, que permita justificar el uso de un determinado Modelos, en este caso especifico se clasificaron según los puntos que se deseen evaluar de un determinado lenguaje. Esta clasificación se dividió entre lenguajes para evaluar la eficiencia y la escalabilidad, pero principalmente para evaluar la capacidad de los Modelos para explotar la definición de restricciones, para el uso de propagadores, para la asignación de valores, y las operaciones sobre los dominios. Dado este análisis se pudo generar un listado de problemas de la CSPLib (librería con CSP) que sirven para probar varios de estos puntos. Entre estos se seleccionó n-reinas y cuadrados mágicos, ya que permiten evaluar una mayor cantidad de puntos, ahorrando tener que modelar varios problemas.

Finalmente debemos hacer referencia la trabajos que no se han podido incorporar y se consideran trabajo futuro, ya que aun hay variados criterios de comparación que no fueron incorporados. Como por ejemplo establecer más grupos de criterios, con mas indicadores que permitan adaptar mejor la evaluación a las necesidades del evaluador, incorporar la opinión de expertos, realizar pruebas de usabilidad, permitiendo así tomar una decisión segura de que lenguaje es mas conveniente para trabajar para resolver CSP. No fueron incorporados, dado que este es un primer acercamiento a generar un plantilla de comparación parametrizable.

Bibliografía.

[1] Federico Barber, Miguel Salido. "Introducción a la programación de restricciones. Inteligencia Artificial", Revista Iberoamericana de Inteligencia Artificial 20, 13–30, 2003.

[2] Roman Barták. "On-line guide to constraint programming", Prague, 1998.

[3] Krzysztof Apt. "Principles of constraint programming", Cambridge University Press, 2003.

[4] E. Tsang, "Foundations of Constraint Satisfaction", Academic Press, London, 1993.

[5] Peter Zoeteweij. "Composing Constraint Solvers. Printed and bound by PrintPartners Ipskamp", Enschede, 2005.

[6] P. Hentenryck. "Constraint Satisfaction in Logic Programming", The MIT Press, 1989.

[7] J. Csontó, J. Paralic. "A look at CLP: theory and application", Applied Artificial Intelligence, 1997.

[8] J. Jaffar, J. Lassez. "Constraint logic programming". Proc. of the 14th ACM Symposium on Principles of Programming Languages (POPL'87), pp. 111–119, 1987.

[9] J. Jaffar, M. J. Maher. "Constraint logic programming: a survey". The Journal of Logic Programming 19,20: pp. 503–581, 1994.

[10] Proc. of the Sixth International Conference on the Practical Application of Prolog and the Fourth Internacional Conference on the Practical Application of Constraint Technology, (PAPPACT98), Publisher Practical Application Company, 1998.

[11] ECLiPSe. User Manual. ECRC, Munich, 2010.

[12] G. Smolka. "The Oz programming model". In Computer Science Today, LNCS 1000, Springer Verlag, pp. 324–343. 1996.

[13] Ilog SOLVER, Reference Manual, version 3.1. 1995.

[14] P. Codognet, D. Diaz. "Compiling constraints in clp". The Journal of Logic Programming 27:185-226, 1996.

[15] T. Fruhwirth. "Constraint handling rules". In Constraint Programming: Basics and Trends, LNCS 910, Springer Verlag, pp. 90–107, 1994.

[16] SICStus Prolog. User's manual, "By the Intelligent Systems Laboratory, Swedish Institute of Computer Science", 1996.

[17] IF/Prolog V5.0A, constraints package. Siemens Nixdorf Informationssysteme AG, Munich, Germany, 1994.

[18] N-F. Zhou. "B-Prolog User's Manual (Version 2.1)". Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka, Japan, 1997.

[19] M. Henz.. "Don't be puzzled!", Workshop on Constraint Programming in conjunction with the 2nd International Conference on Principles and Practice of Constraint Programming (CP'96), 1996.

[20] H. Vandecasteele. "Constraint logic programming: applications and implementation". PhD thesis, Catholic University of Leuven, 1999.

[21] Vipin Kumar. “Algorithms for constraint-satisfaction problems: A survey”. AI Magazine, 13(1):32-44, 1992.

[22] M. R. Garey, D. S. Johnson. “Computers and intractability: A guide to the theory of NP-completeness”. W.H. Freeman and Co, 1979.

[23] Renate Beckmann, Ulrich Bieker, Ingolf Markhof. “Application of constraint logic programming for VLSI CAD tools”. In Constraints in Computational Logics, pages 183-200, 1994.

[24] Ludwig Krippahl, Pedro Barahona. “Applying constraint programming to protein structure determination”. In Principles and Practice of Constraint Programming CP'99, LNCS, pages 289-302, 1999.

[25] Philippe Charman. “Solving space planning using constraint technology”. In Manfred Meyer, editor, Constraint Processing: Proceedings of the International Workshop at CSAM'93, St Petersburg, July 1993

[26] George Dantzig. “Programming in a linear structure”, Econometrica, 1948.

[27] Jean-Louis Lauriere. “A language and a program for stating and solving combinatorial problems”, Artificial Intelligence, 10:29-127, 1978.

[28] ILOG. Ilog solver, reference manual, August 2000.

[29] ILOG. Ilog solver, user's manual, August 2000.

[30] J. Irvin, Lustig, Jean-Francois Puget. “Program does not equal program: Constraint programming and its relationship to mathematical programming”, Interfaces, 31:29-53, December 2001.

[31] Alan K. Mackworth. “Consistency in networks of relations”. Artificial Intelligence, 8:99-118, 1977.

[32] R. Bartak. “Constraint programming: In pursuit of the holy grail”. In Proceedings of the Week of Doctoral Students (WDS), Prague, Czech Republic. 1999

[33] F. Manya, C. Gomes. “Técnicas de resolución de problemas de satisfacción de restricciones”. Inteligencia Artificial, Revista Iberoamericana de IA 7, 169–180, 2003.

[34] U. Montanari. “Networks of constraints: Fundamental properties and applications to picture processing”. Inf. Sci, 7, pp 95–132, 1974.

[35] P. Prosser. “Hybrid algorithms for the constraint satisfaction problem”. Computational Intelligence 9, pp.268-299, 1993.

[36] C. Beck, P. Prosser, R. Wallace. “Toward understanding variable ordering heuristics for constraint satisfaction problems”. In: Fourteenth Irish Artificial Intelligence and Cognitive Science Conference – AICS, pp 11–16, 2003.

[37] B. Smith, P. Sturdy. “Value ordering for finding all solutions”. In Kaelbling, L.P., Saffiotti, eds: IJCAI, Professional Book Center, pp.311–316, 2005.

[38] J. Borrett, E. Tsang, N. Walsh. “Adaptive constraint satisfaction: The quickest first principle”. In Wahlster, ed.: ECAI, John Wiley and Sons, Chichester , pp 160–164, 1996.

[39] F. Benhamou, A. Colmerauer. “Constraint logic programming: selected research”. The MIT Press, Cambridge, MA, 1993.

[40] K. Marriot, P. J. Stuckey. “Programming with constraints”. The MIT Press, Cambridge, Massachusetts, 1998.

[41] R. Dechter. “Constraint processing”. Morgan Kaufmann publisher, 2003.

[42] J.W. Lloyd. “Foundations of logic programming”. Springer-Verlag, Berlin, Heidelberg, 1987.

[43] J. Cohen. “A view of the origins and development of Prolog”. Communications of the ACM, January 1988.

[44] A. Colmerauer, P. Roussel. “The birth of Prolog”. ACM SIGPLAN Notices as part of 2nd ACM SIGPLAN Conference on History of Programming Languages, Cambridge, United States, March, 1993.

[45] Javier Larrosa, Pedro Meseguer. “Algoritmos para satisfacción de restricciones”. Inteligencia Artificial. Revista Iberoamericana de IA, 2003.

[46] M. Dincbas, H. Simonis, P. Van Hentenryck. “Solving large combinatorial problems in logic programming.” Journal of Logic Programming,, pp:75-93, January-March, 1990.

[47] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. “The constraint logic Programming Language CHIP”. In Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, December, 1988.

[48] C. Fierbinteanu. “Constraint Logic Programming Approach of Network Flow Problems Within a Decision Support System Generator for Transportation Planning”, International Journal on Artificial Intelligence Tools, Vol.7, No.4, pp.453-462, 1998.

[49] S. Curtis, B. Smith, A. Wren. “Constructing driver schedules using iterative repair”. In 2nd Internacional Conference on The Practical Applications of Constraint Technology and Logic Programming (PA-CLP'2000), pages 59-78, Manchester, UK, April, 2000.

[50] H. Simonis. “Applications of constraint logic programming”. In L. Sterling, editor, 12th International Conference on Logic Programming (ICLP'95), pages 9-11, Tokyo, Japan, June, 1995.

[51] P. Van Hentenryck, V. Saraswat, Y. Deville. “Design, implementation and evaluation of the constraint language cc(FD)”. In Constraint Programming: Basics and Trends, LNCS 910, Springer Verlag, pp. 293–316, 1996.

[52] P. Codognet, D. Diaz. “Compiling constraints in clp(FD)”. The Journal of Logic Programming 27. pp. 185–226, 1996.

[53] T. Fruhwirth. “Constraint handling rules”. In Constraint Programming: Basics and Trends, LNCS 910, Springer Verlag, pp. 90–107, 1994.

[54] C Schulte. “Solver–An Oz search debugger”. Proceedings of International Workshop on Oz Programming (WOz’95), Martigny, Switzerland, 1995.

[55] IF/Prolog V5.0A, constraints package. Siemens Nixdorf Informationssysteme AG, Munich, Germany, 1994.

[56] T. Muller, J. Wurtz. “Interfacing propagators with a concurrent constraint language”. Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Languages, 1996.

[57] A. Fernandez, P. Hill, “Boolean and Finite Domain solvers compared using Self Referencial Quizzes”. In Proc. Of Joint Conference on Declarative Programming, Italy, 1997.

[58] H. Ait-kaci, A. Podelski. “Towards a meaning of LIFE”. Journal of Logic Programming, in [136], pp.255-274, 1993.

[59] H. Ait-kaci, P. Lincoln, R. Ñasr. “Le Fun: logic, equations and functions”. In Symposium on Logic Programming (SLP'87), San Francisco, California, 1987.

[60] H. Ait-kaci, R Ñasr. “LOGIN: A logic programming language with built-in inheritance”. Journal of Logic Programming, 1986.

[61] Web Oficial GeCode proyect. http://www.GeCode.org

[62] The GeCode team. GeCode/J, a Java interface for GeCode. http://www.GeCode.org/GeCodej.

[63] Choco web site, http://choco-solver.net

[64] C. Schulte,G. Tack. “Views and iterators for generic constraint implementations”. In Recent Advances in Constraints, LNCS, pages 118–132.Springer, 2006.

[65] C. Schulte. “Oz Explorer: A visual constraint programming tool”. Proceedings of the Fourteenth International Conference on Logic Programming, pp. 286–300, The MIT Press Leuven, Belgium, July 1997.

[66] Francois Laburthe. “CHOCO: implementing a CP kernel”. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, Francois Laburthe, Eric Monfroy, Tobias Muller, Laurent Perron, and Christian Schulte, editors, Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000, number TRA9/00, pp.71–85, Singapore, September 2000.

[67] Pascal Brisset, Hani El Sakkout, Thom Fr¨uhwirth, Warwick Harvey, Micha Meier, Stefano Novello, Thierry Le Provost, Joachim Schimpf, and Mark Wallace. “ECLiPSe Constraint Library Manual 5.8. User manual”, IC Parc, London, UK, February, 2005.

[68] Intelligent Systems Laboratory. SICStus Prolog user’s manual, 3.12.1. Technical report, Swedish Institute of Computer Science, Kista, Sweden, April, 2005.

[69] R. Haralick, G. Elliot. “Increasing tree search efficiency for constraint satisfaction problems”. Artificial Intelligence 14: pp 263–313, 1980.

[70] Librería de problemas de satisfacción de restricciones. http://www.csplib.org/

[71] Krzysztof Apt, Mark G. Wallace. “Constraint Logic Programming Using Eclipse”. Cambridge University Press, 2007.

[72] D. Warren. “An Abstract Prolog Instruction Set.” Technical Note 309, SRI, October, 1983.

[73] Arnold, Gosling. “El lenguaje de Programación Java”. Addison-Wesley Iberoamericana, 1997.

[74] Ricardo Soto, Laurent Granvilliers. “The Design of COMMA: An Extensible Framework for Mapping Constrained Objects to Native Solver Models”, ICTAI, 2007.

[75] Cheadle, Harvey, Sadler, Schimpf, Wallace, “A Tutorial Introduction Eclipse”, 2010

[76] Brisse, El Sakkout, Harvey , Wallace, “ECLiPSe Constraint Library Manual”, 2010.

[77] C. Schulte, G. Tack, “Manual Modeling and Programming with Gecode”, 2010.

[78] Eclipse website, http://www.eclipse-clp.org/

[79] Licencia MIT, http://www.opensource.org/licenses/mit-license.php.

[80] Antonio Fernandez, Patricia M. Hill, “A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains”. Springer Netherlands, 2000.

[81] E. Bowring, M. Tambe, M. Yokoo, "Optimize My Schedule but Keep It Flexible: Distributed Multi-Criteria Coordination for Personal Assistants", AAAI Spring Symposium on Persistent Assistants: Living and Working with AI,Menlo Park, CA, March, pp.21-23, 2005.

[82] C. Castro, S. Manzano. "Variable and value ordering when solving balanced academic curriculum problem". In: Proc. of the ERCIM WG on constraints, Prague, June 2001.

[83] Luc De Raedt. Tias Guns. Siegfried Nijssen. "Constraint programming for Itemset mining", KDD conference, USA, 2008.

[84] Jean-Francoise Puget. "Constraint Programming next challenge: Simplicity of use", LNCS Springer Berlin / Heidelberg, 2004.