Máster universitario de Investigación en
Ingeniería del Software y Sistemas informáticos
UNED
Itinerario de Ingeniería de Software: 105128

Análisis automático de modelos de configuración: el sistema operativo eCos

Autor: Beán Castelló, Roberto
Directores: Heradio Gil, Rubén y Fernández Amorós, David

Curso: 2013/2014
Defensa: Junio 2014

Máster universitario de Investigación en
Ingeniería del Software y Sistemas informáticos
UNED
Itinerario de Ingeniería de Software: 105128

Análisis automático de modelos de configuración: el sistema operativo eCos

Trabajo específico propuesto por un profesor

Autor: Beán Castelló, Roberto Directores: Heradio Gil, Rubén y Fernández Amorós, David

Autorización

Autorizo a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma

Roberto Beán Castelló

Agradecimientos

Al apoyo incondicional y compresión de Lydia.
Al apoyo técnico y moral de Rubén Heradio y David Fernández.

Resumen

Se ha implementado una herramienta de análisis automático de las opciones de configuración de la SPL del sistema operativo eCos con el objetivo de reducir el número de respuestas que debe dar el usuario, utilizando la entropía de cada opción de configuración. Esta se calcula a partir de la probabilidad de que se cumpla cada opción de configuración. La probabilidad se calcula transformando el lenguaje de definición de estas opciones de eCos (CDL) a lógica proposicional y luego posteriormente transformando estas expresiones en un BDD. El resultado obtenido es que el uso de un BDD no es la mejor opción para el análisis automático de SPL con muchas opciones de configuración y sin una estructura arbórea bien definida como eCos.

Palabras claves: Análisis automático SPL, BDD, CDL, Configuración de SPL, eCos, Entropía, Ordenación opciones SPL y Probabilidad opciones de SPL.

Índice general

Índice de figuras

Índice de tablas

Índice de listados

Capítulo 1

Introducción

1.1. Contenido

En este capítulo primero se describirá el problema abordado por este trabajo, posteriormente se enumerarán algunos trabajos relacionados, las tareas realizadas y finalmente la organización de los capítulos de este trabajo.

1.2. Configuración sistemas complejos

La configuración de sistemas complejos, como el sistema operativo eCos o *Linux*, requiere que el usuario de respuesta a miles de opciones de configuración.

La selección de una opción de configuración en este tipo de Software Product Line (SPL) hace que se activen o desactiven otras opciones, sin que el usuario a priori conozca cuáles son sus implicaciones.

En el sistema operativo eCos la configuración del sistema de ficheros FAT, requiere la activación previa de 6 opciones de configuración. Una de estas opciones `CYGPK_ISOINFRA` es requisito de otras 52 opciones de configuración. Por tanto, es razonable pensar que primero habría que activar aquellas opciones que influyen en mayor número de opciones.

Teniendo en cuenta lo dicho anteriormente, el orden influye en el número de respuesta que tiene que realizar un usuario. Ya que dependiendo de la opción seleccionada el número de opciones de configuración varía sustancialmente, por tanto, tal y como indican Rubén Heradio y otros en [HFAN$^+$13] el orden tiene una gran influencia en el número de respuestas que debe dar el usuario.

En [HFAN$^+$13] Rubén Heradio y otros proponen utilizar el concepto de *entropía* [Sha48] asociada a la variable aleatoria que representa cada opción de configuración para su ordenación. La *entropía* se calcula a partir de la probabilidad de las variables (opciones de configuración). Para ordenar las opciones de configuración, es necesario calcular la probabilidad de cada una de las opciones.

El objetivo de este trabajo es calcular la probabilidad de cada una de las opciones de configuración del sistema operativo eCos, para verificar en un sistema complejo como eCos la propuesta de ordenación de [HFAN$^+$13].

El cálculo de la probabilidad se realizará pasando las opciones de configuración de eCos a lógica proposicional. A partir de esta lógica proposicional se calculará la probabilidad codificando las expresiones en un BDD.

1.3. Trabajo relacionados

Algunos trabajos anteriores como [CE11], [NE12], [Ste80] y [Ino04] se han interesado en la optimización de la generación de SPL con un número pequeño de características variables, pero no sobre una SPL con un una variabilidad del orden de magnitud de eCos.

La semántica formal del lenguaje CDL de configuración de eCos ha sido estudiada por Berger y otros en [BS10], que realizaron un estudio de la variabilidad de dicho lenguaje en [BSL+12].

Nuestra propuesta propone la optimización de la generación de SPL en un sistema complejo como eCos.

1.4. Tareas a realizar

Para conseguir el objetivo de este trabajo era necesario realizar un sistema que calculara a partir del lenguaje de definición de configuraciones del sistema operativo eCos la probabilidad de cada una de sus opciones de configuración. Para ello ha sido necesario realizar las siguientes tareas:

1. Construir un analizador del lenguaje de configuración del sistema operativo eCos, generando su representación en memoria.
2. Definir las reglas de conversión de la configuración del sistema operativo eCos a lógica proposicional.
3. Transformar el modelo de memoria obtenido en el punto 1 a lógica proposicional, utilizando las reglas de conversión descritas en el punto 2.
4. Convertir la lógica proposicional a un BDD.
5. Calcular la probabilidad de cada una de las variables del BDD.

1.5. Organización del trabajo

En el trabajo primero se describirá brevemente el lenguaje de configuración de eCos CDL (Capítulo 2), luego se presentará la solución propuesta (Capítulo 3), los resultados (Capítulo 4), y finalmente se presentarán las conclusiones obtenidas y futuros trabajos (Capítulo 5).

Capítulo 2

CDL el lenguaje de configuración de eCos

2.1. Contenido

En este capítulo se analizará el lenguaje de configuración del sistema operativo eCos, CDL. Para ello, primero se describirá brevemente eCos, luego CDL y finalmente la estructura de directorios de eCos.

2.2. eCos

eCos, es un sistema operativo de código abierto para aplicaciones embebidas, de respuesta en tiempo real, altamente configurable.

eCos permite configurar al usuario la arquitectura sobre la que se ejecutará, los paquetes y componentes necesarios del sistema operativo, así como las opciones de cada uno de estos paquetes y componentes. Los componentes se organizan tal y como muestra la imagen 2.1.

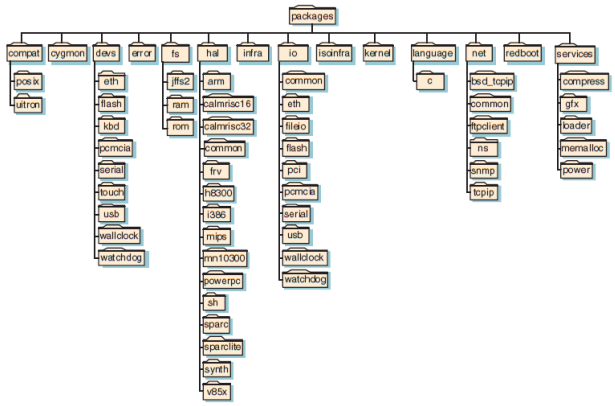


Figura 2.1: Organización a alto nivel de los componentes de eCos[Mas02]

Los componentes del paquete `hal`, a excepción del paquete `common`, identifican a cada una de las arquitecturas sobre las cuales eCos puede ejecutarse.

2.3. CDL

CDL es el lenguaje que eCos utiliza para describir y configurar los paquetes que lo componen. CDL está escrito utilizando TCL. CDL es un DSL embebido [Fow05], ya que está utiliza en el propio lenguaje TCL.

2.3.1. Comandos CDL

El lenguaje CDL implementa cuatro comandos, que definen cada una de las opciones de configuración de eCos:

- `cdl_package`, permite definir un paquete, es decir un componente que puede ser distribuido.
- `cdl_component`, permite definir un componente, una colección de opciones de configuración.
- `cdl_option`, permite definir opciones de configuración.
- `cdl_interafce`, permite definir una opción que puede ser proporcionada por diferentes componentes.

Cada una de las opciones de configuración tiene un nombre asociado que la identifica. Los comandos se ejecutan sobre estas opciones tal y como se muestra en el listado 2.1, en el que se define el paquete `CYGPKG_INFRA` mediante el comando `cdl_package`.

```
cdl_package CYGPKG_INFRA {
}
```

Listado 2.1: Nombre de las opciones

2.3.2. Propiedades de CDL

Cada uno de los paquetes, componentes, opciones e interfaces tienen asociadas una serie de propiedades. Estas propiedades dan información de las características de la opción de configuración como el texto que debe mostrarse al usuario y otras propiedades definen los valores que puede tomar la opción de configuración, cuando esta estará activa o no, que requisitos tiene cada opción o que interfaces implementa, es decir determinan las condiciones en que las opciones de configuración están activas. En esta sección no describiremos todas las propiedades de CDL si no únicamente aquellas que modifican la configuración de eCos. Se puede encontrar una descripción exhaustiva de las propiedades en [Vee01].

- `requires`, la opción de configuración requiere que la expresión indicada esté activa.
- `active_if`, la opción de configuración que contiene esta propiedad queda seleccionada si se está activa la condición indicada en la propiedad.

- `implements`, la opción de configuración implementa la interfaz.
- `parent`, permite definir una estructura jerárquica de configuraciones.
- `flavor`, permite indicar el rango de valores que tiene una opción de configuración.

El listado 2.2 muestra una opción de configuración en la que se indica que para que el componente `COMPONENT1` esté activo se requiere que también este activo el elemento `REQUIRES1`.

```
cdl_component COMPONENT1 {
        requires REQUIRES1
}
```

Listado 2.2: Ejemplo `requires`

Las propiedades también pueden incluir en su sintaxis sentencias simples TCL. Estas sentencias pueden ser evaluadas a cierto o falso. El listado 2.3 muestra un ejemplo, en él se indica que el componente `COMPONEN1` estará activo, si los elemento `REQUIRES1` o `REQUIRES2` están activos y además `REQUIRES3` tiene valor 2.

Las expresiones pueden utilizar los operadores lógicos, de relación, aritméticos y algunas funciones de TCL. Berger y She en [BS10] propusieron una gramática para la definición de estas expresiones. Esta gramática es un subconjunto de la gramática de expresiones de TCL.

```
cdl_component COMPONENT1 {
        requires REQUIRES1 || REQUIRES2
        requires REQUIRES3 == 2
}
```

Listado 2.3: Ejemplo `requires` con sentencias TCL

2.4. Distribución de eCos

Figura 2.2: Directorios de la distribución de eCos

La distribución de eCos genera la estructura de directorios que muestra la figura 2.2 en la que encontramos los siguientes subdirectorios:

doc: Documentación de eCos.

acsupport: Ficheros de soporte de configuración.

examples: Ejemplos de aplicaciones desarrolladas para el sistema operativo eCos.

packages: Código fuente del sistema operativo eCos. El código fuente también incluye la definición de los ficheros de configuración.

tools: Utilidades de eCos.

La estructura del directorio `packages` es idéntica a la organización de componentes presentada en la figura 2.1. Para cada uno de los paquetes individuales, como por ejemplo el paquete del sistema de archivos FAT, se crea un directorio, que para la versión 3.0b1 de eCos se encuentra en la siguiente ubicación `packages\fs\fat\v3_0b1`. Este contiene el directorio `cdl` (figura 2.3), donde se encuentra la configuración del paquete definida mediante su correspondiente fichero CDL.

Figura 2.3: Directorios de un paquete de eCos

Capítulo 3

Propuesta de solución

3.1. Contenido

En este capítulo se detallará la propuesta de solución, para ello primero se describirá su arquitectura (Sección 3.2), y posteriormente cada uno de los módulos que componen la componen (Secciones 3.3, 3.4, 3.6 y 3.7) y las conversiones realizadas (Sección 3.5).

3.2. Arquitectura del sistema

3.2.1. Contenido

En esta sección se describirá la arquitectura del sistema. Esta a partir de una definición de configuración CDL, calcula la probabilidad de cada una sus opciones de configuración. Para ello, primero se presentará el diseño de la arquitectura, y posteriormente se describirá brevemente cada uno sus componentes.

3.2.2. Diseño

La arquitectura del sistema es una arquitectura de *Pipes and filters* (Tubos y filtros) [GS94], se ha divido en diferentes componentes, cada uno de éstos genera una salida, que sirve de entrada para el siguiente componente, tal y como se muestra en la figura 3.1.

Figura 3.1: Arquitectura *Pipes and filters*

3.2.3. Componentes

El diagrama de componentes de la figura 3.2 muestra los diferentes elementos que contiene el sistema y sus dependencias. A continuación describiéremos cada uno de ellos.

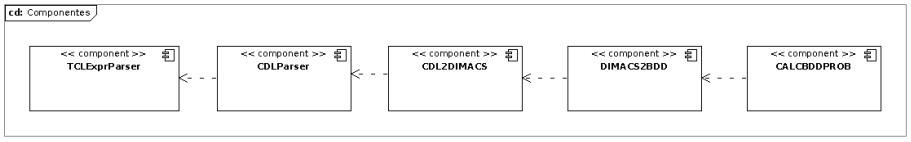


Figura 3.2: Componentes del conversor

CLDParser: Programa TCL que genera una representación en memoria de la definición de una configuración CDL. Recibe como entrada un directorio en el que se encuentran las fuentes del sistema operativo eCos, y genera la representación en memoria de los ficheros de configuración CDL que este contiene.

TCLExprParser: Programa Java que analiza una expresión TCL descrita por [BS10] y genera su representación en memoria. Recibe como entrada una expresión TCL y su contexto y devuelve como salida las instrucciones TCL necesarias para generar la representación en memoria de la expresión.

CDL2DIMACS: Programa TCL que transforma la representación en memoria de una configuración TCL a expresiones en lógica proposicional. Recibe como entrada la representación en memoria de CDL y genera como salida ficheros **DIMACS** extendidos que representan la transformación de la configuración a lógica proposicional.

DIMACS2BDD: Programa C++ que transforma los ficheros **DIMACS** extendidos generados por el componente **CDL2DIMACS** y construye un BDD utilizando el paquete **BuDDy**. Recibe como entrada los ficheros **DIMACS** extendidos y genera como salida un fichero que representa un BDD en **BuDDy**.

CALCBDDPROB: Programa C++ que calcula la probabilidad de cada uno de los nodos de un BDD. Recibe como entrada un fichero con un BDD de **BuDDy** y genera como salida el cálculo de la probabilidad de cada uno de los nodos.

3.3. CDLParser: Analizador del lenguaje CDL

3.3.1. Contenido

En esta sección se describirá la implementación del analizador del lenguaje CDL. Para ello, primero se mostrará su diseño y cómo se ha implementado, posteriormente se describirá cómo se representa en memoria el lenguaje CDL y finalmente cómo se ha reducido el número de opciones de configuración a analizar.

3.3.2. Diseño

Como se ha indicado en la sección 2.3, CDL es un lenguaje desarrollado a partir de TCL, por lo que hubiera implicado la construcción de un analizador de este lenguaje. Para reducir tiempo de desarrollo, se optó por implementar el analizador directamente en el lenguaje TCL. Los comandos CDL son comandos TCL. Por lo que para implementar el analizador se sobrescribieron estos comandos, haciendo que su ejecución genere la representación en memoria de la configuración CDL.

3.3.3. Sobrescritura de comandos

El comando `cdl_component` es un comando TCL válido por lo que sobrescribiendo este comando se puede realizar un analizador del lenguaje CDL. El listado 3.1 muestra la implementación realizada de este comando. En la definición del comando se indica que éste recibe como parámetro el nombre del componente (`name`) y el contenido del comando son sus componentes, opciones y propiedades que este contiene (`content`).

```
1  proc cdl_component {name {content {}}} {
2          addComponent $name [StackCurrent]
3          StackPush $name
4          eval $content
5          StackPop
6  }
```

Listado 3.1: Sobrescritura del comando `cdl_component`

La línea 2 de la ejecución del comando añade el componente con nombre `name` a la representación en memoria y también le pasa como parámetro el padre del componente. El padre del componente lo devuelve el comando `StackCurrent` que retorna el componente anterior que se ha invocado. Una vez añadido el componente se indica que el componente actual es él que se está procesando añadiéndolo a una pila mediante el comando `StackPush`. La instrucción `eval` que se invoca en la línea 4 en TCL hace que el intérprete de TCL ejecute el *content* que se le pasa al comando. Si invocáramos el comando para el listado 2.3 lo que haría sería invocar los comandos `requires`, pasándole como padre el comando `name`. Una vez evaluado el *content* se saca el componente de la pila.

3.3.4. Recorrido por la estructura de directorios

Como se indicó en la sección 2.4 los ficheros de la configuración CDL están distribuidos en cada uno de los paquetes que conforman la estructura de directorios. Por tanto, para analizar todas las configuraciones posibles es necesario analizar todos los ficheros CDL que contienen la estructura de directorios de definición de los paquetes. De esta tarea se encarga los comandos `listfiles` y `parse_file` que muestra el listado 3.2. Para analizar todos los ficheros, lo que se realiza es ejecutar cada uno de los ficheros, con una determina extensión, `.cdl` en nuestro caso.

```
1  source stack.tcl
2
3  proc listfiles { start ext {exclude ""} } {
4          set nodes {}
5          catch { set nodes [glob $start] }
6          foreach node $nodes {
7                  if {[file isfile $node]} {
8                          if { [file extension $node] == $ext } {
9                                  parse_file $node
10                         }
11                 } else {
12                         if { $node != $exclude } {
13                                 listfiles "$node/*" $ext $exclude
14                         }
15                 }
16         }
17 }
18
19 proc parse_file { name } {
20         set content ""
21         set fs [open $name r]
22         set content [read $fs]
23         close $fs
24         #we need set current working directory in the file
25         FileStackPush $name
26         eval $content
27         FileStackPop
28 }
```

Listado 3.2: Comandos `listfiles` y `parse_file`

3.3.5. Representación en memoria de CDL

Representación de objetos en TCL

TCL no es un lenguaje orientado a objetos, por lo que no es posible representar en memoria la definición de una configuración CDL mediante objetos. Para simular objetos Brent B. Welch en [Wel00] propone el uso de vectores globales para definir las propiedades tal y como se muestra en el listado 3.3.

```
1  proc Emp_AddRecord {id name manager phone} {
2          global employee
3          set employee(id,$name) $id
4          set employee(manager,$name) $manager
5          set employee(phone,$name) $phone
6          set employee(name,$id) $name
7  }
8  proc Emp_Manager {name} {
9          global employee
10         return $employee(manager,$name)
11 }
```

Listado 3.3: Objetos en TCL

Ejemplo de representación de objetos

En nuestro caso lo que hemos hecho es utilizar una variable global denominada `features`, esta variable contiene todas las características de las diferentes opciones de configuración del modelo de datos. Una de las características es identificar cuáles son los padres de una determinada opción de configuración tal y como se muestra en el listado 3.4. Para ello, se añaden dos comandos `addFeatureParents` y `getFeatureParents` para establecer los padres y obtenerlos. En este caso `$features(parents,$id)` nos retorna la lista de todos los padres de una opción de configuración.

```
1  proc addFeatureParents { id parent } {
2          global features
3          if { ![featureExcluded $id] && ![featureExcluded $parent] } {
4                  if { [lsearch [getFeatureParents $id] $parent] == -1 } {
5                          lappend features(parents,$id) $parent
6                  }
7          }
8
9  }
10 proc getFeatureParents { id } {
11         global features
12         if { ![info exists features(parents,$id)] } {
13                 return {}
14         } else {
15                 return $features(parents,$id)
16         }
17 }
```

Listado 3.4: Ejemplo de representación de objetos

3.3.6. Reducción del número de configuraciones

El número de configuraciones posibles del sistema operativo eCos son del orden de miles, por lo que es recomendable reducir su número, para hacer que el BDD no sea muy grande. Con este objetivo se han implementado dos estrategias que reducen el número de posibles opciones de configuración.

Selección de la arquitectura

Una de las opciones que el usuario puede decidir es la arquitectura sobre la que se ejecutará el sistema operativo. Esta selección, aunque es concerniente a la elección del usuario, no es algo que este pueda decidir, puesto la arquitectura viene dada por el *hardware*. Por tanto, para reducir las opciones de configuración se eliminan las opciones que no formen parte del hardware seleccionado.

Las opciones dependientes del hardware se encuentran identificadas en la estructura de directorios de eCos como se describió en la sección 2.2, en el directorio `hal`. Existe una carpeta para cada una de las arquitecturas sobre las que se puede ejecutar eCos.

Al seleccionar una arquitectura, se seleccionan las opciones de configuración que sólo sean propias de la arquitectura. Es decir, que no aparezcan en ningún otro fichero de configuración CDL, de otras arquitecturas.

Esta misma estructura dependiente de la arquitectura, también se encuentra en los dispositivos. La configuración de estos dispositivos, se encuentra en el directorio `devs`, por lo que para estos componentes, también se realiza el mismo proceso.

Opciones de configuración `data`

Algunas opciones de configuración tiene asociado el `flavor` (tipo de dato) `data`. En estos casos el usuario debe especificar un valor. Algunos de estos valores no afectan a la configuración de eCos, ya que no aplican restricciones a otras opciones, como por ejemplo `CYGDAT_IO_SERIAL_TTY_TTY0_DEV` que permite indicar cual el dispositivo físico para el terminal 1. Por tanto, estas opciones de configuración son eliminadas para así reducir su número.

3.4. TCLExprParser: Expresiones TCL

3.4.1. Contenido

En esta sección se describe la implementación del conversor de expresiones TCL. Para ello, primero se detallará su necesidad, posteriormente se describirá la herramienta utilizada en la implementación, la gramática del lenguaje, como se transformaran estas expresiones y finalmente la implementación realizada.

3.4.2. Necesidad

Cómo se ha indicado en la descripción de las propiedades CDL algunas de ellas pueden activarse o desactivarse tomando como parámetro de activación una sentencia TCL. Por lo que es necesario, analizar la sentencia TCL para identificar las opciones de configuración que intervienen en la sentencia y sus interrelaciones. Por tanto, se precisa un analizador de las sentencias TCL utilizadas en las expresiones.

En este caso no podemos utilizar la técnica de sobrescritura de los comandos por qué como se ha visto anteriormente las sentencias utilizan directamente operadores que no pueden ser sobrescritos en el caso del lenguaje TCL. Por tanto, es necesario realizar un analizador que sea capaz de convertir una sentencia TCL a su equivalente de representación en memoria.

3.4.3. Herramienta de desarrollo

Se optó por utilizar como analizador el *framework* para el análisis de lenguajes ANTLR V3 (http://www.antlr3.org/). El uso de esta herramienta reduce el tiempo de implementación, ya que a partir de la definición de una gramática BNF realiza acciones para cada símbolo de la gramática reconocido permitiendo así construir la representación en memoria del lenguaje analizado.

3.4.4. Gramática

Tal y como se indicó en la sección 2.3.2, Berger y She propusieron una gramática para las expresiones TCL que utiliza CDL [BS10], que determinaron utilizando ingeniería inversa. Por lo que se optó por partir de esta gramática y realizar un analizador basado en ella.

Se tuvo que modificar la gramática propuesta por Berger y She [BS10], puesto que si bien la gramática reconocía el lenguaje utilizado por CDL, no generaba expresiones correctas, puesto que no se había definido correctamente la precedencia de los operadores de TCL, y que se muestra a continuación:

$$e ::= \text{id} | \texttt{const} | e \otimes e | !e | e \oplus e | e \oslash e | \text{Func}(e, e, ...) | e ? e : e$$

Donde:

- $\otimes \in \{||, \&\&, implies, eqv, xor\}$.
- $\oplus \in \{+, -, *, /, \%, <<, >>, \&, |\}$,
- $\text{Fun} \in \{get\_data, is\_active, is\_enabled, is\_loaded, is\_substr, is\_xsubstr, version\_cmp\}$,

- $\oslash \in \{==, !=, <, >, <=, >=\}$,

id es un identificador de una opción CDL y `const` es una constante entera, de coma flotante o una cadena de texto.

Para corregir el problema se ha utilizado una gramática basada en las expresiones de C extraída de [Ker88], que se muestra en la figura 3.3.

```
⟨expression⟩ ::= ⟨constant_expression⟩ +

⟨constant_expression⟩ ::= ⟨conditional_expresion⟩

⟨conditional_expression⟩ ::= ⟨logical_or_expression⟩ ( '?' ⟨constant_expression⟩ ⟨conditional_expression⟩ ) ?

⟨logical_or_expression⟩ ::= ⟨logica_and_expression⟩ ( OR_OPERATOR ⟨logica_and_expression⟩ )*

⟨logica_and_expression⟩ ::= ⟨equality_expression⟩ ( AND_OPERATOR ⟨equality_expression⟩ ) *

⟨equality_expression⟩ ::= ⟨relational_expression⟩ ( EQUAL_OPERATOR ⟨relational_expression⟩ ) *

⟨relational_expression⟩ ::= ⟨additive_expression⟩ ( RELATIONAL_OPERATOR ⟨additive_expression⟩ ) *

⟨additive_expression⟩ ::= ⟨multiplicative_expression⟩ ( ADITIVE_OPERATOR ⟨multiplicative_expression⟩ ) *

⟨multiplicative_expression⟩ ::= ⟨unary_expression⟩ ( MULT_OPERATOR ⟨unary_expression⟩)*

⟨unary_expression⟩ ::= '!' ⟨primary_expression⟩
    | ⟨primary_expression⟩

⟨primary_expression⟩ ::= ⟨simple⟩
    | ⟨direct_declarator⟩
    | '(' ⟨expression⟩ ')'

⟨direct_declarator⟩ ::= FUNC '(' ⟨expr_list⟩ ')'

⟨simple⟩ ::= ID
    | ⟨constant_expr⟩

⟨constant_expr⟩ ::= STRING
    | INT
    | FLOAT
    | HEX
```

Figura 3.3: Gramática BNF expresiones TCL en CDL

3.4.5. Transformaciones de expresiones

Esta cuestión se describirá en la conversión del modelo en memoria CDL a lógica proposicional.

3.4.6. Implementación

El conversor se ha desarrollado en Java utilizando el *framework* ANTLR. La aplicación Java (`tclexprparser`) recibe como parámetro un fichero que contiene la expresión TCL a transformar, el padre de la expresión a transformar, el tipo de opción de configuración y devuelve la construcción de la expresión en memoria en lenguaje TCL.

3.5. Transformaciones de CDL a lógica proposicional

3.5.1. Contenido

En esta sección se describirá la transformación del lenguaje CDL a lógica proposicional.

3.5.2. Transformaciones

La tabla 3.1 nos muestra cómo realizar las transformaciones del lenguaje CDL a lógica proposicional.

Expresión	**Transformación**
parent `cdl_package` child	$child \rightarrow parent$
parent `cdl_component` child	$child \rightarrow parent$
parent `cdl_option` child	$child \rightarrow parent$
parent `cdl_interface` child	$child \rightarrow parent$
parent `requires` child	$parent \rightarrow child$
parent `active_if` child	$child \rightarrow parent$
parent `implements` child	$parent \rightarrow child$

Tabla 3.1: Transformación de configuración CDL a lógica booleana

Los comandos `cdl_package`, `cdl_component`, `cdl_option` y `cdl_interface` se utilizan para organizar de forma jerárquica las opciones de configuración de CDL. Es decir, que para que esté activo el hijo debe estar activo el padre. Por tanto, debe cumplirse el siguiente predicado $child \rightarrow parent$.

La propiedad de configuración `requires`, implica que para que esté activo el padre, el hijo debe cumplirse. Es decir, la siguiente expresión $parent \rightarrow child$ debe ser cierta.

La propiedad de configuración `active_if` indica que al cumplirse el hijo automáticamente el padre queda activado. Por tanto, debe cumplirse la expresión $child \rightarrow parent$. Esta configuración a diferencia de las anteriores asegura que si el hijo esté activo el padre a su vez este activo.

La propiedad de configuración `implements` indica que la opción de configuración implementa la opción indicada en el comando. Por tanto, en caso de que se cumpla el padre se deberá cumplir el hijo, es decir la interfaz que implementa. Por tanto, esta restricción se consigue con la expresión $parent \rightarrow child$.

3.6. CDL2DIMACS: Conversor de CDL a lógica proposicional

3.6.1. Contenido

En esta sección se detallará cómo se ha realizado la conversión de la representación en memoria de CDL obtenida por **CDLParser**, a lógica proposicional. Para ello, primero se detallará la salida del módulo, posteriormente cómo se realizó la primera transformación infructuosa, luego se describirá las características del problema a resolver, la heurística de ordenación de la conversión utilizada, y finalmente la implementación realizada.

3.6.2. Salida del módulo

Este módulo debe transformar el lenguaje CDL a lógica proposicional, para pasar esta información al siguiente módulo, por tanto se requería almacenar de algún modo las cláusulas de lógica proposicional. El estándar de facto de intercambio de lógica proposicional es el **DIMACS**, que es utilizado por la mayoría de SAT *solvers* como Minisat o zChaff. Por lo que se decidió optar por este formato.

3.6.3. Transformación inicial

Inicialmente se optó por realizar una transformación directa sin ningún orden preestablecido. Al realizar esta transformación el crecimiento del número de nodos del BDD fue muy rápido provocando que este no fuera tratable, tal y como se mostrará en la sección 4.2.

Cuando se construye un BDD, su tamaño depende fuertemente del orden en que se introduzcan las variables en el BDD. Por tanto, para asegurar que el crecimiento sea menor, es necesario ordenar correctamente las variables [RK08], utilizando una heurística. Esta heurística depende del problema, por lo que era necesario utilizar una adaptada a problemas de configuración de SPL.

3.6.4. Características del problema

Los paquetes eCos tienen una estructura jerárquica y las opciones de configuración están en su mayor parte relacionadas con el paquete que se está configurando. Por ejemplo, las opciones del paquete `CYGPKG_FS_FAT`, forman una estructura jerárquica tal y como se muestra en el listado 3.5.

```
cdl_package CYGPKG_FS_FAT {
        requires CYGPKG_ISOINFRA
        requires CYGINT_ISO_ERRNO
        requires CYGINT_ISO_ERRNO_CODES
        cdl_option CYGNUM_FS_FAT_NODE_HASH_TABLE_SIZE {
        }
        cdl_option CYGNUM_FS_FAT_NODE_POOL_SIZE {
        }
        cdl_option CYGNUM_FS_FAT_BLOCK_CACHE_MEMSIZE {
        }
        cdl_option CYGDBG_FS_FAT_NODE_CACHE_EXTRA_CHECKS {
        }
        cdl_option CYGCFG_FS_FAT_USE_ATTRIBUTES {
        }
        cdl_option CYGPKG_FS_FAT_RET_DIRENT_DTYPE {
        }
```

```
        cdl_component CYGPKG_DEVS_DISK_TESTING {
                cdl_option CYGDAT_DEVS_DISK_TEST_DEVICE {
                }
                cdl_option CYGDAT_DEVS_DISK_TEST_DEVICE2 {
                }
        }
       cdl_option CYGPKG_FS_FAT_TESTS {
       }

}
```

Listado 3.5: Paquete `CYGPKG_FS_FAT`

Cada uno de los paquetes tienes sus propias opciones de configuración. Esto hace que se conformen grupos de opciones (clústeres). Estos clústeres tienen dependencias internas en su mayoría y otras dependencias que forman parte de otros grupos. Por ejemplo en el caso del paquete `CYGPKG_FS_FAT` que requiere de las opciones configuración `CYGPKG_ISOINFRA`, `CYGINT_ISO_ERRNO` y `CYGINT_ISO_ERRNO_CODES` que forman parte del paquete `CYGPKG_ISOINFRA`.

Nina Narodytska y Toby Walsh en [NW07] proponen para este tipo de problemas de configuración una heurística basada en estos clústeres que se forman en los problemas de configuración. Por lo que se optó por utilizar esta heurística, que describiremos a continuación.

3.6.5. Heurística de ordenación de las variables

Para problemas con muchas variables, como es el caso de la configuración de eCos (2562 variables y 1506 restricciones para la arquitectura **i386**, por ejemplo), Nina Narodytska y Toby Walsh proponen en [NW07] el siguiente algoritmo:

Paso 1: Identificar las variables que son fuertemente dependientes y que forman un clúster. Para eso utilizan el algoritmo Markov Cluster Algorithm (MCL) [vD00].

Paso 2: Ordenar las variables dentro de los clústeres de acuerdo a una variable central de la cual dependen el resto de variables. Para ello, proponen el siguiente algoritmo:

2.1 Para cada variable del clúster, calcular el peso de las variables adyacentes. Refiriéndose a este valor como el peso de la variable.

2.2 Seleccionar la variable con mayor peso dentro del clúster. Esta variable es considerada como la variable central del clúster y la primera dentro del orden.

2.3 Las variables en el clúster son ordenadas por el peso de las aristas que conectadas con la variable central del clúster. Las variables que no están conectadas con la variable central son puestas al final del clúster.

2.4 Las variables que no están ordenadas en paso anterior, son ordenadas de acuerdo a sus pesos. Las variables más pesadas primero. Variables con igual peso son ordenadas por el peso total de sus hijos.

Paso 3: Se establece el orden dentro de los clústeres, se ordenan los que están débilmente conectados con el resto de restricciones primero, asumiendo que los clústeres independientemente conectados no incrementan mucho

el tamaño del clúster. El grado de aislamiento de un clúster es calculado de acuerdo a su proyección. La proyección es "La suma de los pesos de las aristas para cada nodo en el clúster (correspondiente a los clústeres vecinos), relativo al peso de todas las aristas de ese nodo". La proyección es "la media de la proyección de todos los nodos del clúster". Los clústeres son ordenados de modo descendiente de acuerdo a su proyección.

A parte de la ordenación de las variables en [NW07], también se propone para minimizar el tamaño del BDD el uso de la ordenación dinámica.

3.6.6. Implementación

Cálculo de los clústeres (MCL)

Al igual que proponen Nina Narodytska y Toby Walsh en [NW07] se utilizado el algoritmo MCL para el cálculo de los clústeres. Para ello se ha utilizado el software **MCL-edge** de Stijn van Dongen y que implementa el algoritmo MCL que él describe en [Don00].

El software **MCL-edge** recibe como entrada una fichero de texto en formato que denomina *abc*, este formato tiene dos etiquetas (la 'A' y la 'B') que identifican a los nodos de las aristas de un grafo y un valor numérico 'C' que identifica lo similares que son ambos nodos del grafo.

El único parámetro de entrada del software **MCL-edge** aparte de la definición de la similitud de los nodos del grafo, es el parámetro `I` (inflación). Este parámetro controla la granularidad del clúster. Es decir, el tamaño de los clústeres identificados, cuanto más pequeño es el parámetro se generan menos clústeres. En [Don], se indica que valores adecuados de este parámetro oscilan en el rango [1.2:5]. Pero el valor, adecuado de este parámetro depende de cada problema.

La figura 3.4 muestra el número de clústeres obtenidos para la arquitectura **i386**, asignado como valor 1 el valor 'C' del fichero de entrada. En esta figura se aprecia que a medida que aumenta el valor de `I` el número de clústeres también aumenta.

El módulo **CDLParser** construye un modelo en memoria de la definición de configuración CDL. Para generar el fichero en formato *abc*, se utiliza el siguiente algoritmo:

Paso 1: Se recorren todas las opciones de configuración encontradas.

- Paso 1.1 Para cada opción de configuración se obtienen todos los elementos que dependen de ella, es decir sus nodos hijos.
- Paso 1.2 Para cada nodo hijo y padre encontrado se calcula la similitud de ambos elementos.

El algoritmo MCL utiliza lo similares que son los nodos de las aristas, para agrupar los elementos del clúster. Por tanto, el parámetro de similitud determina como se agruparan los clústeres. En [NW07] no se indica cual es el valor seleccionado para identificar la igualdad de los nodos.

Las opciones de configuración serán similares si forman parte de un mismo paquete, o bien tienen variables comunes de otros paquetes que hacen que estos estén activos. Es decir, dependerán de la similitud de los elementos de los cuales

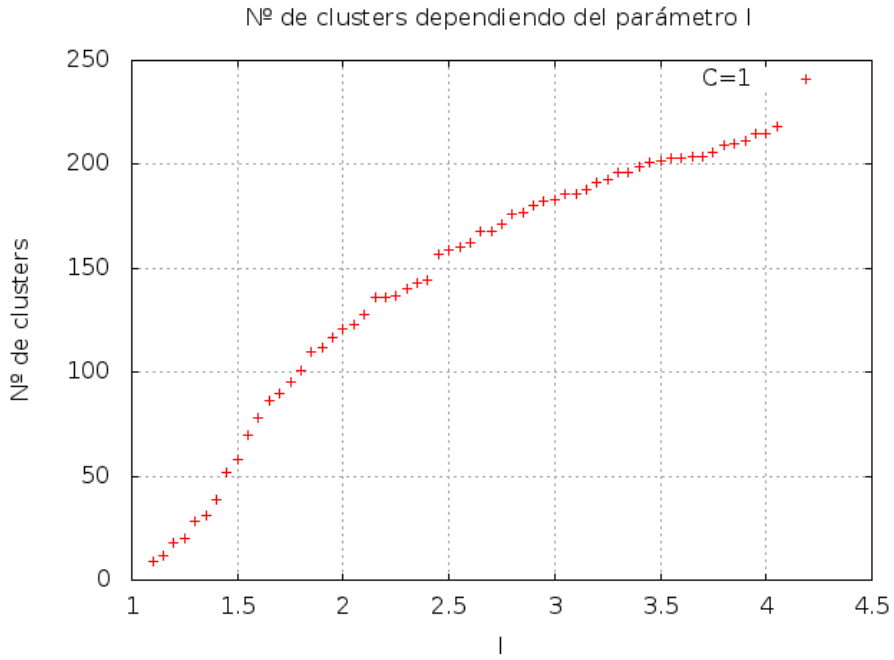


Figura 3.4: Nº de clústeres para la arquitectura **i386** con valor 'C' igual a 1

dependen es por ello que el criterio de similitud seleccionado. Para calcular la similitud de las opciones de configuración se ha utilizado el siguiente algoritmo:

Paso 1: Se calculan todos los nodos padres del primero nodo de arista, de modo recursivo. Se deberá tener en cuenta que un nodo podrá aparecer varias veces en la estructura arbórea puesto que puede tener varias restricciones. Por lo que en el caso, que se haya visitado un nodo este no vuelve a visitarse.

Paso 2: Se calculan todos los nodos padres para el segundo nodo de la arista de igual modo a lo hecho en el `Paso 1`.

Paso 3: Se calcula la similitud como el número de nodos padres comunes que tengan ambos nodos.

Este algoritmo hace que el decrezca del número de clústeres respecto cuando se varía el parámetro `I` con respecto al valor de `C` igual 1, tal y como muestra la figura 3.5. Es decir, agrupar las opciones de configuración, haciendo que disminuya el número de clústeres.

Ordenación de variables y clústeres

Se ha implementado el algoritmo descrito en la sección 3.6.5 en TCL utilizando el modelo de datos que genera el módulo **CDLParser**. El resultado obtenido, son **n** clústeres y dentro de cada uno de los clústeres una ordenación de las variables.

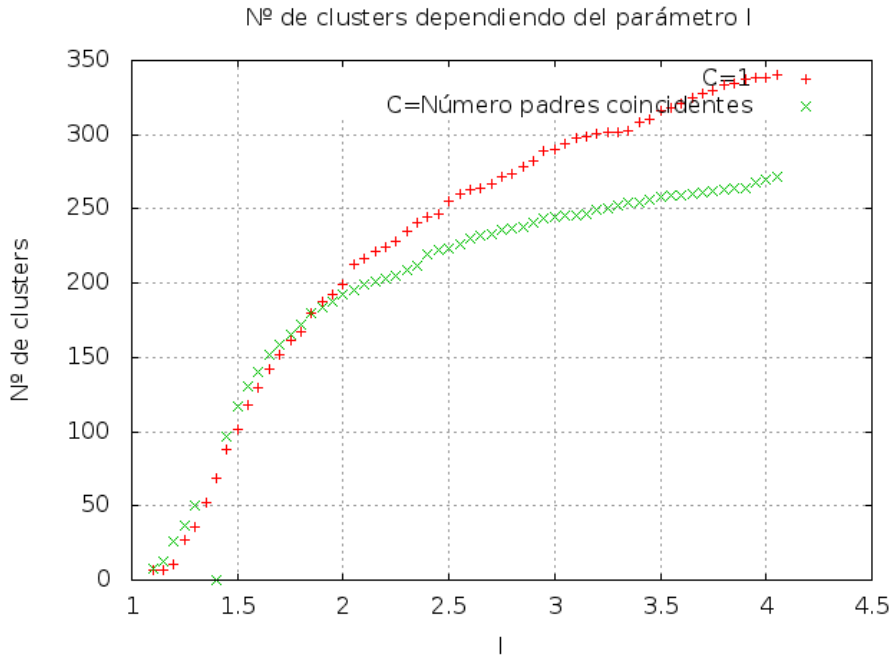


Figura 3.5: Comparativa del número de clústeres dependiendo del valor `C` e `I`

Clústeres a lógica proposicional

Una vez obtenidos los clústeres y la ordenación de sus variables es necesario transformar estas variables a lógica proposicional para ello se sigue el algoritmo que muestra el listado 3.6.

```
1  cluster_actual := 1;
2  mientras cluster_actual <= numero_total_clusters
3          variable := 1;
4          mientras variable <= numero_total_variables_cluster(cluster)
5                  para_cada hijo en elementos_hijos_de_la_variable(variable) hacer
6                          tipo_relacion := obtener_tipo_relacion(variable, hijo);
7                          transformar_restriccion(tipo_relacion, variable, hijo);
8                  fpara_cada
9                  variable := variable + 1;
10         fmientras
11         cluster_actual := cluster_actual + 1;
12 fmientras
```

Listado 3.6: Transformación a lógica proposicional

Hay que comentar que si bien hay elementos que quedan fuera del clúster, incluirlos luego ellos solos tiene un comportamiento peor, ya que está fuera de cualquier relación y al estar ordenados el comportamiento es correcto.

Este algoritmo recorre todos los clústeres y sus variables. Para cada clúster genera un fichero en formato **DIMACS** con las restricciones encontradas, dicho fichero contiene las conversiones descritas en la sección 3.5.2 transformadas a formato **DIMACS** que codifica las formulas en forma normal conjuntiva (CNF, Conjuntive Normal Form) [HR04]:

$$L ::= p|\neg p$$
$$D ::= L|L \vee D$$
$$C ::= D|D \wedge C$$

El listado 3.7 codifica fórmula CNF 3.1 en el formato **DIMACS**. Las líneas que comienzan por el carácter `c` son comentarios, la línea que comienza por el carácter `p` codifica en el primer valor el número de variables y en el segundo valor el número de cláusulas. Cada una de las variables en las cláusulas se codifica con un número entero positivo, y en el caso que esté negado con el número en negativo. Cada una de las líneas son fórmulas *or*, y se realiza una operación *and* con la siguiente línea.

```
c Comentarios
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Listado 3.7: Formato **DIMACS**

$$\begin{aligned} &x_1 \vee \neg x_5 \vee x_4 \\ &\neg x_1 \vee x_5 \vee x_3 \vee x_4 \\ &\neg x_3 \vee \neg x_4 \end{aligned} \tag{3.1}$$

El formato **DIMACS** no permite incluir expresiones TCL como la expresión vista en la sección 2 `REQUIRES3==2`. La codificación de estas expresiones se ha realizado como comentarios. Extendiendo de este modo el formato **DIMACS** para incluir este tipo de expresiones.

3.6.7. Transformación de expresiones a lógica proposicional

En la gramática definida en la sección 3.5.2 hemos identificado dos tipos de expresiones:

1. Aquellas que utilizan operador lógicos (`&&`,`||`, `!`).
2. Aquellas que utilizan operaciones de relación (`<`,`>`,`>=`, `<=`, `=`, `!=`)y aritméticas (`+`, `-`, `*`, `/`, `%`).

Las primeras pueden codificarse en CNF directamente, transformando las expresiones proposicionales a CNF utilizando un algoritmo de transformación expresiones proposicionales a CNF como el que se describe en [HR04].

Las segundas no pueden codificarse directamente a CNF, ya que hay una parte de la expresión que no es lógica proposicional, como por ejemplo, la que se muestra en el listado 3.8. Esta expresión `requires`, se transformaría en la siguiente expresión 3.2, utilizando las transformaciones expresadas en la sección 3.5. Esta expresión puede considerarse como un todo que se evaluará a cierto o falso.

```
CYGNUM_FS_FAT_NODE_POOL_SIZE >= (CYGNUM_FILEIO_NFILE+2)
```

Listado 3.8: Formato DIMACS

$$PARENT \rightarrow (\texttt{CYGNUM\_FS\_FAT\_NODE\_POOL\_SIZE} >= (\texttt{CYGNUM\_FILEIO\_NFILE} + 2)) \quad (3.2)$$

Estas expresiones complejas las codificamos en el formato **DIMACS** como comentarios, tal y como se muestra en el listado 3.9. En él se identifica mediante la palabra clave `complex` que se trata de una expresión compleja, y se aisla la expresión compleja entre los caracteres # y luego se codifica la expresión la implicación como *or* para que de este modo el conversor de formato **DIMACS** extendido a BDD, **DIMACS2BDD** pueda identificarlo y transformarlo correctamente.

```
c complex: # v1171 >= v2551 + 2# || -v1171
```

Listado 3.9: Expresiones complejas en DIMACS

3.7. DIMACS2BDD: DIMACS extendido a BDD

3.7.1. Contenido

El esta sección se describirá la implementación de la aplicación que transforma el formato **DIMACS** extendido descrito anteriormente a un BDD. Para ello, primero se realizará una breve descripción de qué es un BDD, luego se describirá el paquete utilizado para generar el BDD (**BuDDy**), y posteriormente se describirá cómo se han realizado las transformaciones del formato **DIMACS** extendido a **BuDDy**, y finalmente, se detallará la salida del módulo.

3.7.2. BDD

En lógica proposicional existe el operador `if-then-else` $(x \rightarrow y_0, y_1)$ definido como:

$$x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

Cualquier expresión en lógica proposicional puede expresarse utilizando el operador `if-then-else` y las constantes 0 (falso) y 1 (cierto). Por ejemplo, $\neg x$ es $(x \rightarrow 0, 1)$ y $x \iff y$ es $x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 0, 1)$.

Un BDD es un modo de representar expresiones de lógica proposicional [And97], mediante un grafo a cíclico con raíz como:

- Uno o más nodos terminales etiquetas con 0 o 1, y
- un conjunto de nodos variables u, con dos nodos hijos. Las dos aristas son dadas por dos funciones $low(u)$ y $high(u)$. La primera de ellas es el resultado de asumir que la variable u tiene valor 0 (representado como una línea discontinua) en una expresión `if-then-else` y el segundo valor 1 (representado como una línea continua).

Por ejemplo el BDD de la expresión $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$ se correspondería al siguiente BDD que muestra la figura 3.6.

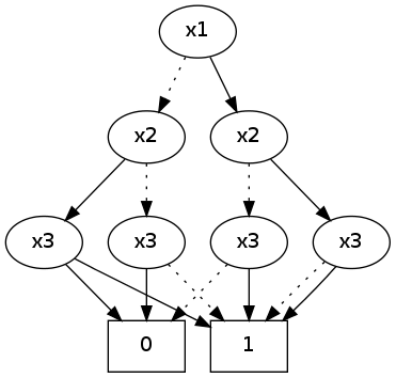


Figura 3.6: BDD expresión $(\neg x_1 \wedge \neg x_2 \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$

Un BDD está ordenado si todos los caminos a través del grafo respetan el orden lineal $x_1 < x_2 \cdots x_n$. Un BDD ordenado está reducido si:

Unicidad: No existen dos nodos distintos u y v que tienen el mismo nombre de variable y los sucesores *low* e *high* iguales es decir:

$$var(u) = var(v) \wedge low(u) = low(v) \wedge high(u) = highv \rightarrow u = v$$

No test redundante: Ningún nodo u tiene valores idénticos de los sucesores *low* y *high*, es decir:

$$low(u) \neq high(u)$$

3.7.3. BuDDy

BuDDy es un paquete implementado en C/C++ que permite generar BDD's. Este paquete también permite:

- Codificar variables enteras como BDD, como un vector de variables booleanas que representan al número.
- Realizar operaciones lógicas (<,>,>=,<=,=,≠) con variables enteras.
- Y realizar operaciones matemáticas con dichas variables (+,−,∗,/,%).
- Ordenación dinámica de variables.

Esta funcionalidad adicional que implementa **BuDDy** permite codificar las expresiones complejas TCL descritas en la sección 3.6.7, y la ordenación dinámica de variables que recomiendan Nina Narodytska y Toby Walsh en [NW07].

3.7.4. Métodos de ordenación dinámica de BuDDy

BuDDy implementa 7 métodos de ordenación dinámica de variables[CWWG], se ha seleccionado el método de ordenación *sift* [Rud93], puesto que es el utilizando en [NW07] y la documentación de **BuDDy**, indica que es un buen método de ordenación aunque lento. El problema en nuestro caso no es un problema de tiempo, si no de crecimiento del árbol, ya que el cálculo se realizará sólo una vez. Por tanto, el tiempo que tarde el BDD en generarse no es una restricción del problema.

3.7.5. Algoritmo de transformación

El algoritmo de transformación a BDD lee todos los ficheros que genera el módulo **CDL2DIMACS**, este módulo genera un fichero **DIMACS** extendido para cada clúster, dándole como nombre del fichero el número del clúster. Por tanto, transformando a BDD cada uno de los ficheros que genera el módulo en orden ascendente se realiza la transformación a BDD. Este proceso lo realiza el algoritmo que se muestra en el listado 3.10.

```
1
2  fichero := obter_primer_fichero_dimacs(directorio);
3  //Se inicializa el bdd con el valor cierto
4  bdd := inicializar_bdd();
5  mientras queden_ficheros_por_tratar(directorio) hacer
6          fichero := obter_fichero_a_tratar(directorio);
7          cnf := obtener_representacion_en_memoria(fichero);
8          para_cada clausula en cnf hacer
9                  si es_simple_clausula(clausula) entonces
10                         bdd := bdd and transformar_clausula_a_bdd(clausula);
11                 sino
12                         bdd := bdd and transformar_clausula_compleja(clausula);
13                 fsi
14         fpara_cada
15         fichero := obtener_siguiente_fichero(directorio);
16 fmientras
```

Listado 3.10: Algoritmo de transformación

La primera tarea que realiza el algoritmo (línea 1) es inicializar el BDD en el caso de **BuDDy**, se establece el número de nodos máximo, la memoria que reserva la aplicación ambas constantes definidas en el fichero "`consants.h`"del módulo **DIMACS2BDD**. También se indica que el método de ordenación va a ser SIFT. Y se inicializa el BDD con valor cierto, de modo que se puedan ir añadiendo cláusulas.

El algoritmo una vez inicializado el BDD lee cada uno de los fichero que representa un clúster lo analiza y obtiene su representación en memoria (`cnf`). Una vez obtenida su representación en memoria el sistema verifica si la cláusula es una simple (sólo operadores proposicionales) y en caso de ser así realiza su transformación haciendo un `and` con el BDD `bdd`, es decir añadiendo clausulas. O bien transformado la cláusula compleja y realizando la misma operación.

Transformación de cláusulas simples

En este caso la transformación de la cláusula es muy sencilla, se inicializa un BDD temporal (`tmpbdd`) a valor falso y se recorren todas las variables que forman la cláusula y se realiza una operación `or` con dicho BDD negando la variable en caso de que esta esté negada, tal y como muestra el listado 3.11.

```
1  void CNF2BDD::convertSimpleClause(Clause* _clause){
2          bdd tmpbdd=bddfalse;
3          bdd bddvar;
4          Variable* var;
5          for(std::list<Variable*>::iterator it = _clause->getVariables()->begin();
6                  it != _clause->getVariables()->end(); it++ ){
7                  var = (*it);
8                  bddvar = bdd_ithvar(this->getVarNum(var->Number()));
9                  if (var->Negated()){
10                         tmpbdd = tmpbdd | !bddvar;
11                 }else{
12                         tmpbdd = tmpbdd | bddvar;
13                 }
14         }
15         this->m_bdd = this->m_bdd & tmpbdd;
16 }
```

Listado 3.11: Transformación de clausulas simples

Transformación de cláusulas complejas

Como se ha indicado anteriormente **BuDDy** es capaz de transformar a lógica proposicional las variables enteras incluidas en operaciones lógicas y matemá-

ticas, este tipo operaciones son las que incluyen las cláusulas complejas tal y como se indicó en la sección 3.4 TCLExprParser: Expresiones TCL.

Para el uso de variables enteras **BuDDy** utiliza Finite Domain Variables (FDV) que permiten representar números enteros cómo lógica proposicional y construir operaciones lógicas y matemáticas con ellas transformándolas a su correspondiente BDD, utilizando esta funcionalidad de **BuDDy** se genera un bloque de variables finitas para cada expresión compleja. A continuación describiremos como se realizaría por ejemplo la transformación de la siguiente expresión:

```
#v1171 >= v2551 + 2#
```

El primer paso a realizar es identificar el número de variables numéricas que contiene la expresión compleja contenida entre los caracteres #. En este caso, serían dos variables numéricas `v1171` y `v2551`, una vez identificadas el número de variables se construirá el dominio finito para dos variables enteras utilizando la función `fdd_extdomain`, que recibe como parámetro un vector con el rango de cada una de las variables y el número de variables del dominio finito.

Si la expresión incluye constantes numéricas es necesario crear el vector que la represente utilizando la función `bvec_con` que permite representar una constante como un vector booleano.

Tras identificar las variables y constantes bastará con operar con ellas utilizando las operaciones que proporciona **BuDDy**. Primero sumamos 2 a la variable `v2551` utilizando la función `bvec_add` y luego posteriormente el resultado que nos proporcionara esta operación, lo compararíamos con la variable `v1171` utilizando la función `bdd_gth` que nos proporcionará un BDD. Con éste BDD podremos operar entonces de igual modo al que hacemos con las expresiones sencillas.

3.7.6. Resultado del módulo

El módulo **DIMACS2BDD** genera un BDD. Este se graba en un fichero de modo que pueda ser procesado posteriormente. Para ello, se utiliza el comando `bdd_fnsave` de **BuDDy** que permite grabar el BDD en un fichero de texto y que luego puede ser recuperado mediante el comando `bdd_fnload`.

3.8. CALBDDPROB: Cálculo de probabilidad de BDD

3.8.1. Contenido

En este capítulo se describirá el módulo de cálculo de probabilidad de cada una de las variables de un BDD. Este módulo podrá ser utilizado para otros problemas.

3.8.2. Calculo de probabilidad de una opción de configuración

A partir de un BDD se puede obtener la probabilidad de que se produzca cada variable, debido a que podemos contar el número de posibles caminos que hacen que un determinado nodo tenga el valor cierto, utilizando el algoritmo que se describe en [FAR13].

Traduciendo lo expresado en el párrafo podemos extrapolar que si el BDD es la representación del modelo de configuración de eCos. La probabilidad de que un nodo se cumpla, nos dirá la probabilidad de que una determinada opción de configuración pueda ser seleccionada. Por tanto, calculado esta probabilidad tendremos una medida de las opciones que más se darán, es decir aquellas con una probabilidad más alta.

El objetivo de este módulo es calcular las probabilidades de cada una de las opciones de configuración de eCos o de cualquier otro BDD guardado en formato BuDDy.

3.8.3. Implementación

El módulo **CALCBDDPROB** está implementado en C++, este lee el BDD almacenado en un fichero y calcula su probabilidad mediante el comando `bdd_get_probability_rec` tal y como se muestra en el listado 3.12.

```
1  #include <iostream>
2  #include "bdd.h"
3  #include "constants.h"
4  #include "bdd_probability.h"
5
6
7  int main(int argc, char **argv) {
8          BDD bddtocalc;
9          mpq_t *prob_rational;
10         int num_var, num_nodes, i, j;
11         if( argc > 1 ) {
12                 bdd_init(NODENUM,CACHESIZE);
13                 if( !bdd_fnload(argv[1],&bddtocalc) ) {
14                         num_var = bdd_varnum();
15                         num_nodes = bdd_nodecount(bddtocalc)+2;
16                         prob_rational = new mpq_t[num_var*sizeof(mpq_t)];
17
18                         std::cout << "# P(g) ##########################\n";
19                         bdd_get_probability_rec(bddtocalc , prob_rational,
20                                                 num_var, num_nodes, num_nodes-1);
21                         for (i=0; i<num_var;i=i++){
22                             gmp_printf("P( %d) = %Qd\n", i, prob_rational[i]);
23                         }
24                 }
25                 bdd_done();
26         }else{
27                 std::cerr<<"calcbddprob bdd_file"<<std::endl;
28         }
29 }
```

Listado 3.12: Implementación `calcbddprob`

Capítulo 4

Resultados experimentales

4.1. Contenido

En este capítulo se presentan los resultados experimentales obtenidos. Cómo afectan los parámetros `I` (inflación) y `C` igualdad de los nodos al crecimiento del número de nodos del BDD, y ejemplos en los cuales nuestro algoritmo ha sido capaz de generar el BDD.

4.2. Crecimiento del número de nodos

Como se ha indicado anteriormente el mayor problema de los BDD es el crecimiento del número de nodos, que es exponencial, tal y como muestra la figura 4.1 en la que se observa el crecimiento del BDD, realizando una transformación sin agrupar las características en clústeres y sin ninguna ordenación de variables.

El crecimiento del número de nodos del BDD se amortigua aplicando el algoritmo de [NW07] tal y como se muestra en la figura 4.2, donde se compara el crecimiento del número de nodos sin agrupar (figura 4.1). El crecimiento que se visualiza en la figura se ha calculado para un valor de `I` igual a 2 y para un valor `C` igual a 1. Este resultado experimental valida el algoritmo descrito en [NW07].

Pero el algoritmo **MCL-edge** como se ha descrito anteriormente tiene dos parámetros que varían la distribución de los clústeres. El parámetro de inflación `I` y el peso o igualdad que se le da a las aristas, el parámetro `C`. Para evaluar, estos parámetros se ha realizado el siguiente experimento se han realizado variaciones del parámetro `I` de valores 1.05 a 4 y se han implementado tres modos de cálculo del parámetro `C`:

1. Asignar el mismo valor a todas las aristas.
2. Asignar el valor igual al número de padres comunes.
3. Asignar la inversa del valor del número de padres comunes.

Para evaluar, que parámetros son los que permiten un crecimiento más lento del BDD se ha limitado el número de nodos del BDD y se ha ejecutado el proceso

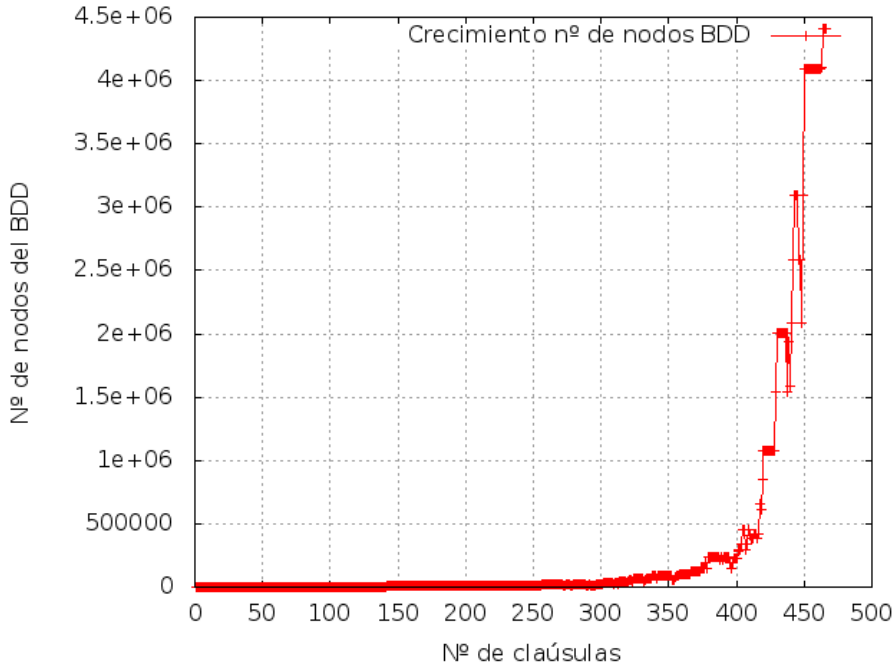


Figura 4.1: Crecimiento nodos del BDD

de conversión del formato **DIMACS** extendido a BDD hasta que se ocupen todos los nodos disponibles, contando el número de restricciones convertidas. Aquellos, parámetros que nos den el mayor número de restricciones convertidas serán aquellos que harán que el crecimiento del BDD sea menor.

La figura 4.3 nos muestra el número de cláusulas convertidas hasta llegar al valor máximo de nodos fijado para el BDD (10.000.000). En ella podemos apreciar que el mayor número de cláusulas lo obtenemos para `I` igual 1.45. Tanto para un valor de `C` igual a 1, como contando el número de nodos padres. Y por tanto, este debería ser el valor que el parámetro `I` debería tomar.

Se ha ejecutado el módulo utilizando este parámetro y otros valores a nivel experimental, pero no ha sido posible calcular el BDD para el problema, ya que aunque el algoritmo [NW07] suaviza el crecimiento del número de nodos es insuficiente para poder procesarlo, ya que **BuDDy** sólo puede direccionar un máximo de 2.147.483.647 nodos, debido al límite de los número enteros en C/C++. A ese límite se llega con un *hardware* con 8 GB de RAM.

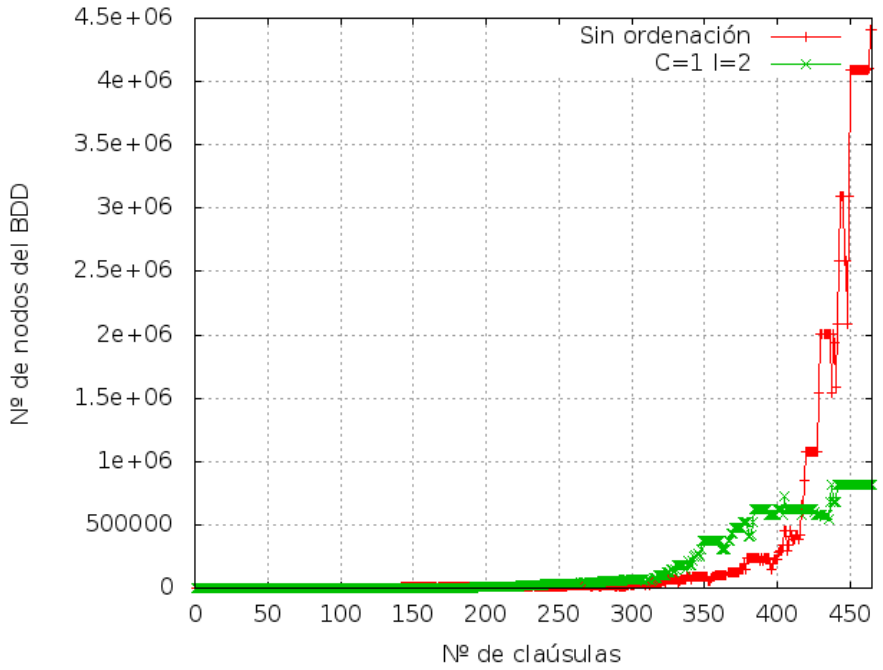


Figura 4.2: Comparativa del crecimiento del número de nodos del BDD

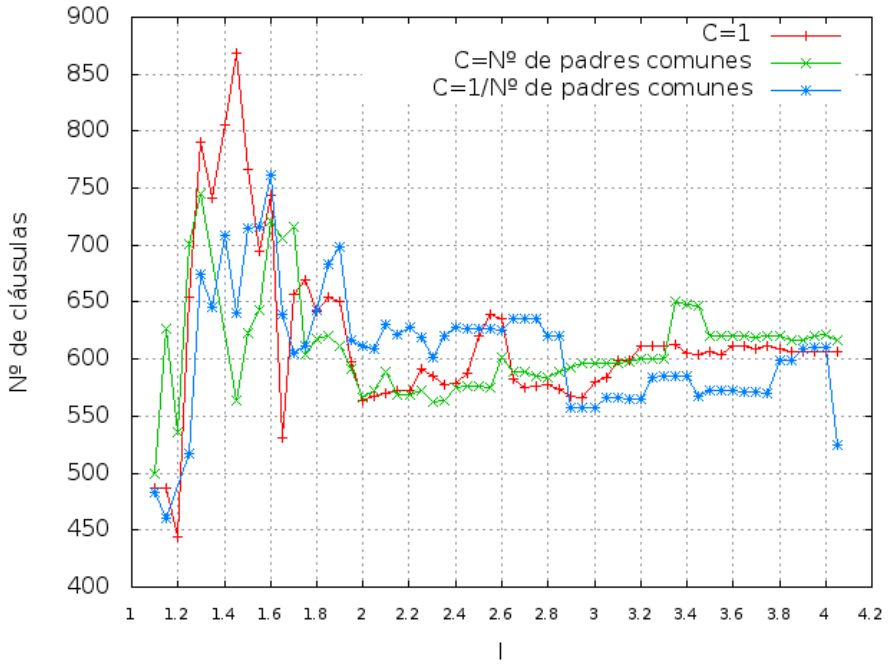
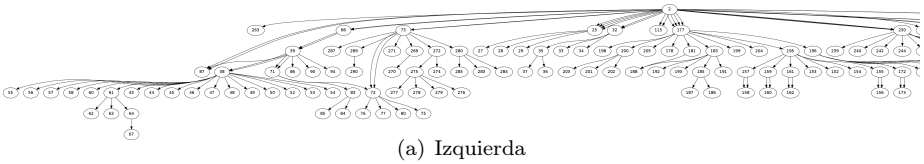


Figura 4.3: Comparativa del crecimiento dependiendo del valor de `C`

4.3. Paquetes individuales de eCos

Hemos sido capaces de construir un BDD para paquetes complejos de eCos, como por ejemplo el `io` (367 opciones de configuración y 277 restricciones), o `net` (409 opciones de configuración y 308 restricciones). Estos tienen una estructura jerárquica. En el caso del paquete `io` totalmente jerárquica, tal y como muestra la figura 4.4.



(a) Izquierda

(b) Derecha

Figura 4.4: Restricciones paquete `io`

Al combinar los paquetes, la estructura ya no tiene una estructura completamente jerárquica ya que algunas opciones generan dependencias en múltiples paquetes. Por ejemplo, la opción de configuración `CYGINT_ISO_ERRNO` que forma parte del paquete `isoinfra`, y afecta a 52 opciones de configuración que forman parte de los paquetes `io`, `fs`, `compact`, `net`, `services` y `language`.

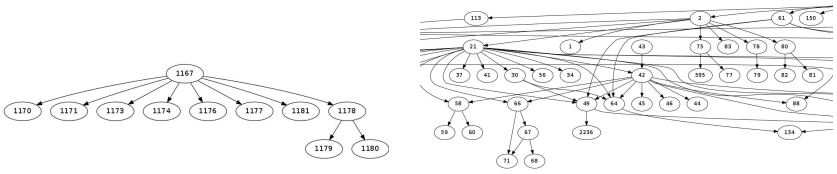
Capítulo 5

Conclusiones y futuros trabajos

5.1. Análisis de eCos con BDD

Tal y como se ha indicado en la sección 4.2 del capítulo anterior, no ha sido posible construir un BDD completo dado el crecimiento exponencial de este, aun utilizando heurísticas adecuadas para problemas de configuración.

Analizando visualmente las dependencias entre variables se observa que el problema de configuración de eCos si bien tiene una estructura arbórea en alguna de sus partes (figura 5.1(a)), otras opciones de configuración no presentan una estructura totalmente arbórea (figura 5.1(b)). Esta característica de eCos puede hacer que las heurísticas de problemas de configuración con estructuras arbóreas no sean aplicables.



(a) eCos estructura arbóreo

(b) eCos parcialmente arbóreo

Figura 5.1: Estructura de opciones eCos

5.2. Heurísticas

Como continuación de la línea de investigación se propone buscar o diseñar heurísticas de ordenación de variables en BDD que se adapten a problemas de configuración como eCos, que no tienen una estructura completamente jerárquica.

5.3. Reducción del problema

Las restricciones de las expresiones TCL con valores numéricos codificadas con variables de dominio finito, introducen más variables adicionales al BDD. Se propone también identificar estas restricciones y codificarlas de forma más eficiente para así reducir el problema.

Bibliografía

[And97] Henrik Reif Andersen, *An introduction to binary decision diagrams*, Tech. report, Course Notes on the WWW, 1997.

[BS10] Thorsten Berger and Steven She, *Formal semantics of the cdl language*, Tech. report, Department of Computer Science, University of Leipzig, Germany, January 2010, Technical Note.

[BSL+12] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki, *Variability modeling in the systems software domain*, Tech. report, University of Waterloo, 2012.

[CE11] Sheng Chen and Martin Erwig, *Optimizing the product derivation process.*, SPLC (Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John, and Klaus Schmid, eds.), IEEE, 2011, pp. 35–44.

[CWWG] Haim Cohen, John Whaley, Jorn Wildt, and Nikos Gorogiannis, *Buddy: Variable reordering*, `http://buddy.sourceforge.net/manual/group__reorder.html`, Accedido Abril 2014.

[Don] Stijn Dongen, *Mcl manual*, `http://micans.org/mcl/`, Accedido Mayo 2014.

[Don00] _____, *A cluster algorithm for graphs*, Tech. report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 2000.

[FAR13] David Fernádez-Amorós and Heradio Rubén, *Computing the probabilities of the variables of a boolean formula using binary decision diagrams*, Tech. report, Spanish Open University, 2013.

[Fow05] Martin Fowler, *Language workbenches: The killer-app for domain specific languages?*, 2005.

[GS94] David Garlan and Mary Shaw, *An introduction to software architecture*, Tech. report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[HFAN+13] Rubén Heradio, David Fernández Amorós, Alexander Nöherer, López-Herrejón Roberto, and Alexander Egyed, *Entropy-based minimization of the number of steps to configure a software product*, Tech. report, Universidad Nacional de Educación a Distancia, 2013.

[HR04] Michael Huth and Mark Ryan, *Logic in computer science: Modelling and reasoning about systems*, Cambridge University Press, New York, NY, USA, 2004.

[Ino04] R.B. Inouye, *Minimizing the length of non-mixied inititative dialogs*, Workshop on the Association for Computionational Linguistic on Studen Research, 2004.

[Ker88] Brian W. Kernighan, *The c programming language*, 2nd ed., Prentice Hall Professional Technical Reference, 1988.

[Mas02] Anthony Massa, *Embedded software development with ecos*, Prentice Hall Professional Technical Reference, 2002.

[NE12] Alexander Nöhrer and Alexander Egyed, *Optimizing user guidance during decision-making*, 15th International Software Product Line Conference, August 2012.

[NW07] Nina Narodytska and Toby Walsh, *Constraint and variable ordering heuristics for compiling configuration problems*, IJCAI-2007 (Hyderabad, India) (Manuela M. Veloso, ed.), January 2007, pp. 149–154.

[RK08] Michael Rice and Sanjay Kulhari, *A survey of static variable ordering heuristics for efficient bdd/mdd construction*, Tech. report, Univeristy of California, Riverside, 2008.

[Rud93] Richard Rudell, *Dynamic variable ordering for ordered binary decision diagrams*, Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design (Los Alamitos, CA, USA), ICCAD '93, IEEE Computer Society Press, 1993, pp. 42–47.

[Sha48] Claude Shannon, *A mathematical theory of communication*, Bell System Technical Journal **27** (1948), 379–423, 623–656.

[Ste80] L. Steinbert, *Question ordering in a mixed initiative program specification dialogue*, 1st Annual National Conference on Artificial Intelligence, August 1980.

[vD00] Stijn van Dongen, *Graph clustering by flow simulation*, Ph.D. thesis, University of Utrecht, Utrecht, May 2000, http://www.library.uu.nl/digiarchief/dip/diss/1895620/inhoud.htm.

[Vee01] Bart Veer, *The ecos component writer's guide*, `http://ecos.sourceware.org/docs-3.0/user-guide/ecos-user-guide.html`, 2001, Accedido Abril 2014.

[Wel00] Brent B. Welch, *Practical programming in tcl and tk (3rd ed.)*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

Apéndice A

Acrónimos

ANTLR ANother Tool for Language Recognition

BDD Binary Decision Diagram

BNF Backus Naur Form

CDL Component Definition Language

CNF Conjuntive Normal Form

DSL Domain Specific Language

eCos Embedded Configurable Operating System

FAT File Allocation Table

FD Feature Diagramas

FDV Finite Domain Variables

GB Gigabyte

MCL Markov Cluster Algorithm

RAM Random Access Memory

SAT Satisfiability

SPL Software Product Line

TCL Tool Command Language