Manual de Arlips 3.1

Grupo de Tecnología Informática - Inteligencia Artificial
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia (España)

v. 3.1.0

Índice

1. Introducción **3**

2. Compilación en Arlips **3**

3. Estructura general de un sistema de producción Arlips **5**

4. Especificación léxica **6**

5. Directivas y macros **7**

6. Tipos de datos **7**
6.1. Tipos *integer* y *real* 7
6.2. Tipo *boolean* 8
6.3. Tipos *intlist* y *realist* 8
6.3.1. Funciones *head* y *tail* 9
6.3.2. Funciones *pophead* y *poptail* 9
6.3.3. Funciones *addhead* y *addtail* 10
6.3.4. Función *size* 10
6.3.5. Manejo de errores 11
6.4. Rangos de valores 11
6.5. Tipo *array* 11

7. Constantes **12**

8. Clases **12**

9. Instancias **14**

10. Variables globales **15**

11. Funciones **17**
11.1. Otras funciones predefinidas 17
11.1.1. print 17
11.1.2. halt 18
11.2. Funciones declaradas por el usuario 18
11.3. Funciones externas 19

12. Estrategia de control **20**

13. Reglas **21**
13.1. Parte izquierda de las reglas 22
13.2. Parte derecha de las reglas 24

14. Variables patrón **25**

15. Depuración de un sistema de producción **27**
15.1. Ejecución del sistema basado en reglas por pasos o con límite 27
15.2. La acción *facts* 28
15.3. Opción de compilación *-log* 28

16. Lista de palabras reservadas **30**

1. Introducción

En los entornos de tiempo real, los lenguajes tradicionales para el desarrollo de sistemas basados en reglas no son adecuados debido a la enorme dificultad que implica su análisis temporal. Por esta razón, se diseñó un nuevo lenguaje, denominado *Arlips* ("A Real-Time Language for Integrated Production Systems"), y su correspondiente algoritmo de *pattern matching*, que cumple las condiciones necesarias para su análisis temporal.

Arlips presenta una sintaxis y semántica similar a la de otros lenguajes basados en reglas que siguen la semántica propia de los sistemas de producción (especialmente "Clips"), pero al mismo tiempo se han incluido restricciones que facilitan que se pueda realizar su análisis temporal. La principal restricción impuesta por el lenguaje Arlips para permitir su análisis temporal es que el tamaño de la memoria de trabajo debe conocerse estáticamente.

El traductor construido para el lenguaje Arlips genera código C para las tres fases de ejecución de un sistema basado en reglas. Arlips dispone de una doble interfaz con el lenguaje C: es sencillo integrar funciones de una biblioteca externa y al mismo tiempo es posible integrar todo el sistema basado en reglas Arlips en otros proyectos de mayor envergadura.

Entre las principales características de Arlips se puede destacar la existencia de una jerarquía de marcos ("frames") con herencia simple, la existencia de variables patrón, prioridad estática y dinámica en las reglas, estrategia de control definida estática o dinámicamente, coerción y operadores sobrecargados, variables globales ajenas al proceso de pattern matching, posibilidad de cuantificar la variables existencial y universalmente, tipos de datos entero, real, booleano y listas, ...

En este manual se presenta una breve introducción al lenguaje Arlips. Se supone que el lector tiene conocimientos de programación en general. Más concretamente, es deseable que el lector posea conocimientos de programación de sistemas basados en reglas que sigan el paradigma de los sistemas de producción usados en Inteligencia Artificial.

2. Compilación en Arlips

El compilador Arlips traduce los sistemas de producción escritos en Arlips a código C, generando distintos ficheros , incluido un "makefile" que permite la compilación de todo el código generado mediante la orden *make*.

El compilador de Arlips y el código fuente que debe compilar deben encontrarse en el mismo directorio para la correcta generación de ficheros. Se recomienda encarecidamente almacenar cada programa en un directorio específico al programa para evitar que se mezcle el código fuente de distintos sistemas basados en reglas. Además, algunos de estos ficheros tienen nombres comunes para programas distintos, como por ejemplo, el fichero *Makefile*.

El traductor Arlips se invoca desde la línea de comandos. Su sintaxis es:

```
  $arlips nombre_fichero.arl [-e:{prof,anch}] [-debug]
[-noprints] [-nomake] [-nomain] [-v] [-input dir] [-output dir]
[-artis] [-log]
```

nombre_fichero.arl es el nombre del fichero en donde se ha escrito el sistema de producción.

Arlips soporta las siguientes opciones:

- **-e:{prof, anch}** especifica la estrategia de control que se usará en el sistema de producción. *prof* indica que el sistema de producción usará una estrategia de control basada en profundidad. *anch* indica que el sistema de producción usará una estrategia de control basada en anchura. Por defecto se utiliza profundidad. Es posible especificar la estrategia de control en el propio sistema de producción, e incluso variarla en tiempo de ejecución. Cualquier estrategia indicada en el código del sistema de producción tendrá preferencia sobre esta opción.

- **-debug** crea un código fuente que facilita el depurado del sistema de producción. Se muestran los cambios que se producen en la base de hechos, las activaciones y desactivaciones de reglas, la ejecución de reglas...

- **-noprints** El código generado no incluirá ninguna salida por pantalla (*printf*), aunque apareciesen entre las acciones del lado derecho de las reglas. Tampoco se incluirá ningún mensaje de error en caso de que se produzca un error en tiempo de ejecución.

- **-nomake** no genera el fichero *Makefile* para la compilación del código C generado por Arlips.

- **-nomain** no se genera la función *main* del código C. Además, el fichero *Makefile* es renombrado al nombre del fichero que contiene el código fuente, pero con la extensión *.mak*. Todo ello facilita la integración del código generado con otro código C, o la combinación de varios sistemas de producción Arlips.

- **-v** El compilador de Arlips imprime por la salida estándar las acciones que está llevando a cabo mientras procesa el sistema de producción.

- **-input *dir*** Por defecto el compilador Arlips espera tener el código a compilar en el directorio desde el que se le llamó. Esta opción permite modificar este comportamiento, estableciendo el directorio en el que se espera el código a compilar a *dir*.

- **-output *dir*** Por defecto el código generado por el compilador Arlips es almacenado en el mismo directorio en donde se encuentra el código

Arlips. Esta opción permite modificar este comportamiento, estableciendo el directorio en el que se generará el código a *dir*.

- **-artis** Indica de manera explicita al compilador Arlips que el código generado debe ser compatible con *Artis*. Arlips es capaz de activar de manera automática el modo de compatibilidad con *Artis* cuando se importa una descripción de *kdm* del mismo[1].

- **-log** Cuando se activa la opción *log* Arlips genera código capaz de registrar la ejecución del sistema de producción, registrando el disparo de reglas y las activaciones y desactivaciones que éstas producen sobre la agenda, así como el estado de la memoria de trabajo en aquellos momentos en los que la ejecución de una regla así lo solicite mediante la orden *facts*.

3. Estructura general de un sistema de producción Arlips

Los sistemas de producción definidos en Arlips se componen de:

- Declaración de constantes.
- Declaración de variables globales.
- Declaración de clases.
- Declaración de instancias.
- Declaración de funciones.
- Declaración de funciones externas.
- Declaración de variables patrón.
- Declaración de reglas.
- Declaración de estrategia de control.

Cada una de estas partes queda delimitada por paréntesis. El orden en que se producen las distintas declaraciones no es relevante siempre y cuando cualquier objeto usado haya sido previamente declarado. Por ello es recomendable declarar las variables patrón tras las clases e instancias, y las reglas tras las variables patrón.

[1] Consulte la directiva *#import kdm* en la sección 5

4. Especificación léxica

Los identificadores usados deben cumplir ciertas restricciones:

- Por defecto, solo son significativos los 30 primeros caracteres. Los identificadores pueden ser más largos, pero Arlips solo considerará los 30 primeros. Debe prestarse especial atención a este particular cuando se usen funciones externas. Si los nombres de las funciones externas tienen mayor longitud, las llamadas a las mismas no se generarán de forma adecuada, y el código C generado no compilará. En caso de ser necesario, el tamaño máximo de los identificadores podría ajustarse, pues viene definido por la constante *long_ident* en el fichero *constant.h* del código fuente del compilador.

- Todos los identificadores deben comenzar con una letra.

- Siempre y cuando cumplan la condición anterior, los identificadores pueden ser cualquier secuencia de letras minúsculas (a..z) y mayúsculas (A..Z), dígitos (0..9) y subrayados, que no formen una palabra reservada.

- Arlips no distingue entre mayúsculas y minúsculas con respecto a las palabras reservadas del lenguaje. Sin embargo, Arlpis si distinguirá entre mayúsculas y minúsculas en el resto de los casos; esto incluye los identificadores que declare el usuario, las cadenas de literales o los nombres de las funciones externas.

- La lista de palabras reservadas se encuentra en la última sección de este manual.

- Las cadenas usadas para la salida estándar pueden tener una longitud máxima por defecto de 255 caracteres. El tamaño máximo de las cadenas de salida viene definido por la constante *max_cadena_comillas* en el fichero *constant.h* y puede ser modificado en caso de ser necesario.

- La longitud de la lista de parámetros de las funciones externas no debe superar los 255 caracteres.

- La lista de parámetros de las funciones externas debe coincidir con la de una declaración *extern* de la función en C.

- Los comentarios van encerrados entre llaves {}. También se considerará un comentario cualquier secuencia de caracteres precedida por un punto y coma "*;*" y hasta el final de la línea.

5. Directivas y macros

#include "fichero" permite la inclusión de archivos dentro de otros. Al llegar a una directiva include se abre y compila el fichero indicado, tras ello se continúa el procesado del fichero que contenía la directiva *#include*.

#extfile "fichero" indica a Arlips en qué ficheros de código objeto se podrían encontrar las funciones externas declaradas. Arlips no realiza comprobación alguna en dichos ficheros, ni prueba la existencia de las funciones, pues esa tarea pertenece al linkador de C. Esta directiva solo genera las referencias necesarias en el fichero *Makefile* para que el código C compile y se genere el enlace adecuadamente.

#import kdm "fichero.arl" hace que se importe un fichero que contenga una definición de un kdm de *artis*. Las definiciones de kdm de *artis* se realizan en un *fichero.**xml*** que debe haber sido procesado por los parser de *artis*, generando el *fichero.**arl***. Arlips se comportará como si se hubiese realizado un *#include* de *fichero.arl* con unas cuantas salvedades que a continuación se detallan.

Arlips creará código condicional, capaz de funcionar tanto si el código escrito en arlips es una *ks crítica* (con restricciones temporales) u *opcional*. Arlips no generará el fichero *Makefile*. El su lugar se generan dos fichero con extensión *.mak* que facilitarán la compilación del código c generado. El fichero *nombre_de_programa.mak* permitirá la compilación del código como si se tratase de una tarea opcional. Por su parte el fichero *rt_nombre_de_programa.mak* permite la compilación del código cuando éste sea usado como una tarea crítica.

Finalmente, arlips habrá generado dos funciones adicionales que permitirán al principio de la ejecución del SBR la lectura de todos los datos del kdm que vaya a usar el SBR, para así obtener sus valores más recientes. De la misma manera, cuando el SBR vaya a finalizar su ejecución volcará en el kdm el estado actual (desde la perspectiva del SBR) de todos los datos que haya usado.

6. Tipos de datos

Arlips ofrece algunos tipos de datos que pueden ser usados para definir instancias, constantes, variables o atributos de clase. Estos tipos de datos nativos al lenguaje son enteros (*integer*), reales (*real*), *tipos rango de valores enteros*, lógicos (*boolean*), listas de enteros (*intlist*), listas de reales (*realist*) y vectores *array* de cualquiera de los tipos simples (enteros, reales y lógicos).

6.1. Tipos *integer* y *real*

La palabra reservada para indicar un tipo entero es *integer*. Para un tipo real es *real*.

El rango de valores enteros y reales soportado en tiempo de ejecución, dependerá del compilador C y la máquina que se emplee para compilar y ejecutar el código C generado por el compilador de Arlips[2].

Los operadores aritméticos suma "+", resta "-", multiplicación "*" y división "/.^{es}tán sobrecargados y por lo tanto se emplean con operandos enteros y reales. Además, el compilador de Arlips maneja la conversión automática de enteros a reales (coerción).

6.2. Tipo *boolean*

Cualquier elemento de tipo lógico puede tomar los valores *true* o *false*. La representación de dichos valores al realizar la transformación a C es la habitual: *false* es representado por el valor entero *0* y *true* por *1*. Las operaciones admitidas entre elementos de tipo lógico son *and* y *or*. Es posible, aunque no recomendable, combinar estos operandos con elementos de tipo entero y real, con la misma semántica de C.

6.3. Tipos *intlist* y *realist*

La palabra reservada *intlist* se usa en la declaración de variables globales, instancias y slots de una clase, para indicar que el identificador correspondiente, usando la sintaxis adecuada que más adelante se explica, es de tipo lista de números enteros. Por su parte, la palabra reservada *realist* se usa para declarar listas de números reales.

Los tipos lista de enteros y lista de reales admiten por defecto hasta 255 elementos en sus listas, pero se puede especificar explícitamente el tamaño máximo de las listas. Para ello, tras la declaración del tipo, se debe encerrar entre corchetes el valor del tamaño máximo de la lista. La declaración del tamaño máximo de la lista prevalece sobre el valor por defecto del tamaño máximo de lista y sobre la inicialización de la lista. Si no se produce tal declaración de tamaño máximo de la lista, para establecer el tamaño máximo de la lista se usa como segundo criterio la inicialización de la misma que se puede hacer opcionalmente en la declaración. Cuando tal inicialización existe, el tamaño de esa lista inicial será el que marque el tamaño máximo de la lista. Solo cuando no se produce ninguna de las dos declaraciones anteriores se usa el valor por defecto. Aunque aun no se conozca la sintaxis de la declaración de instancias, veamos algunos ejemplos:

```
(l1 of intlist)
(l2 of intlist (1 2 3))
(l3 of intlist[9])
(l4 of intlist[9] (1 2 3))
```

[2]En la implementación actual del compilador de Arlips, los reales se imprimen con 6 decimales.

l1, *l2*, *l3* y *l4* son cuatro declaraciones distintas de instancias de tipo lista. La lista *l1* no se inicializa ni se le declara un tamaño máximo, por tanto, podrá contener hasta 255 elementos. La lista *l2* se inicializa con una lista de 3 elementos y al no estar declarado ningún tamaño máximo de lista, se asume que dicho tamaño máximo es el de la lista de inicialización, esto es, 3. Aunque la lista *l4* sí se inicialice, también se declara un tamaño máximo de lista, tal y como se hace con la lista *l3*, por tanto, el tamaño máximo de ambas listas es el declarado, en este caso 9.

Las listas de reales están sujetas a las mismas restricciones de precisión que los números reales: cualquier valor entero que sea declarado como real se transformará a un valor real con la misma precisión que ofrezca la función *fprintf* de C para variables de tipo *float*.

Arlips proporciona las funciones de consulta y manipulación de listas que se presentan en la siguiente sección.

6.3.1. Funciones *head* y *tail*

La función *head* permite consultar el primer elemento de la lista, devolviendo su valor. Por su parte, la función *tail* devuelve el último elemento de la lista. Ambas funciones pueden ser usadas tanto en la parte izquierda como en la parte derecha de las reglas. Dada la lista *l*, la consulta de su cabeza y su último elemento podría hacerse como sigue:

```
(head(l))
(tail(l))
```

Es posible asignar el valor devuelto, compararlo para crear una condición o imprimirlo, tal y como se haría con cualquier valor numérico del mismo tipo de la lista. Suponiendo todas las variables o instancias correctamente declaradas, las reglas pueden contener:

```
...
;Parte izquierda de la regla
(head(l)>valor) ; comparación
...
;Parte derecha de la regla
(cabeza:=head(l)) ; asignación
...
(print("El último elemento de l es ", tail(l))) ; Impresión por pantalla.
```

6.3.2. Funciones *pophead* y *poptail*

Estas funciones devuelven respectivamente el primer y último elemento de la lista, borrándolo de la misma. Aunque el lenguaje no lo prohíbe ex-

plícitamente, no se recomienda el uso de estas funciones en la parte izquierda de las reglas, pues ambas funciones modifican la base de hechos.[3]

Si *l* es una lista no vacía y *cabeza* y *cola* son variables o instancias compatible con el tipo de los elementos de la lista, las siguientes llamadas dentro de una regla asignan a *cabeza* y *cola* el primer y el último elemento de la lista *l* respectivamente, y los borra de la lista.

```
(cabeza:=pophead(l))
(cola:=poptail(l))
```

6.3.3. Funciones *addhead* y *addtail*

Estas funciones añaden un nuevo elemento al inicio o final de la lista respectivamente. La sintaxis de uso es la que sigue:

```
(addhead(exp l))
(addtail(exp l))
```

donde

- *exp* es una expresión simple de tipo compatible con el tipo de *l*.
- *l* es un identificador de lista. El identificador de lista puede ser un slot de una instancia. Para ello se usa la notación *instancia.slot*. Es posible el uso de variables patrón cuantificadas existencialmente siempre y cuando la instanciación que produzcan hagan referencia a una lista.

Dado que estas funciones modifican la memoria de trabajo, deben ser usadas en la parte derecha de las reglas de forma análoga a *pophead* y *poptail*.

6.3.4. Función *size*

Esta función devuelve el número de elementos de una lista dada en un instante. *size* puede ser usado tanto en la parte izquierda como en la parte derecha de las reglas. Su sintaxis es:

```
(size(l))
```

donde *l* es un identificador de lista o una variable patrón cuantificada existencialmente.

[3]No se recomienda la utilización de estas funciones en la parte izquierda de la regla, ya que producen un efecto colateral (modificación de la memoria de trabajo) que debería realizarse siempre en el lado derecho. Al mismo tiempo, hay que recordar que solo las modificaciones de la memoria de trabajo que se producen en la parte derecha de las reglas provocan la reevaluación de las condiciones de las reglas y son capaces de activar o desactivar reglas.

6.3.5. Manejo de errores

En tiempo de ejecución pueden suceder errores relacionados con las funciones que consultan o modifican las listas. Por ejemplo, no es posible extraer un elemento de una lista que esté vacía. De la misma forma, tampoco se puede consultar la cabeza ni la cola, ni se pueden añadir nuevos elementos a una lista que ya esté llena. Arlips proporciona un manejo de errores en tiempo de ejecución. Con el fin de facilitar la adaptación de Arlips a cualquier tipo de sistema, dicho control de errores se ha implementado en una función del fichero generado *tipos_.c* llamada *error_lista*. Esta función imprime información por la salida de errores acerca de la llamada y la lista que provocan el fallo. Si se desea, el contenido de esta función puede ser modificado para adaptarlo a cualquier sistema de manejo de errores.

6.4. Rangos de valores

Arlips permite declarar slots de clases o variables como rango de enteros[4]. Un rango de valores enteros es el conjunto de números enteros comprendidos entre dos enteros, y que incluye a ambos. Para declarar que la variable o el slot es de un rango de valores determinado se usa la faceta *type*, indicando el menor y el mayor de los números del rango, separados por dos puntos "..". Por ejemplo:

```
...
(slot rango (type 3..5))
...
```

indicaría que el slot llamado *rango* solo puede tomar valores comprendidos entre el *3* y el *5*.

El compilador Arlips comprueba la corrección del tipo de manera estática, pero no genera el código necesario para garantizar la corrección del valor de forma dinámica. El usuario es responsable de dicha tarea.

6.5. Tipo *array*

Arlips soporta vectores de los tipos elementales *integer*, *real* y *boolean*. Para la declaración de un vector se ha reservado la palabra *array*. Tras ello se debe indicar el tamaño del vector mediante la notación de corchete clásica de C. Finalmente debe seguir la palabra reservada *of* seguida del tipo básico del vector. En la actualidad, los tipos básicos de vector soportados son *integer*, *real* y *boolean*. El acceso a los elementos del vector también sigue la notación clásica de C, considerándose el *[0]* el primer elemento del vector.

[4]No es posible realizar esto con instancias.

7. Constantes

Las constantes son identificadores cuyo valor asignado no cambia durante la ejecución del sistema de producción. Arlips solo soporta constantes de tipo entero y real. Su declaración se realiza usando la siguiente sintaxis:

```
(defconst
   (cte_i valor_i)
   (cte_j valor_j)
   ...
)
```

Como se puede observar, es posible declarar simultáneamente varias constantes usando para ello una lista de pares *(atributo valor)*. El tipo de una constante no se declara explícitamente. Éste vendrá dado por el tipo del valor al que se declara la constante. El siguiente ejemplo muestra como declarar dos constantes; *pi* es una constante real a la que se le asigna el valor *3,14159*, *elem* es una constante entera que toma el valor *10*.

```
(defconst (pi 3.14159) (elem 10) )
```

Una constante puede ser usada en los mismos lugares donde pueda usarse un valor numérico del mismo tipo de la constante, como por ejemplo, el la declaración del valor por defecto de instancias, o en los argumentos de llamadas a funciones.[5]

8. Clases

Arlips permite la utilización de marcos "frames" que permiten estructurar la representación del conocimiento del sistema basado en reglas. Para ello el programador puede definir "clases" entre las que se permite la herencia simple. La sintaxis para la declaración de una clase es la siguiente:

```
(defclass cl isa cl_padre
   (slot s1 (type t1) (default v1))
   (slot s2 (type t2) (default v2))
    ...
)
```

donde

- *cl* es el nombre de la clase

[5]Las declaración de constantes Arlips son traducidas por el compilador a sentencias *#define* en el código C generado, pero al mismo tiempo, el compilador Arlips sustituye los nombres por sus valores en el código C generado.

- isa *cl_padre* es una parte opcional usada para declarar que la clase *cl* hereda de su clase padre *cl_padre*. De esta manera se puede establecer una jerarquía de clases.

- *s1*, *s2* son identificadores de los distintos slots de la clase. Cada slot puede verse como un atributo de la clase.

- *t1*, *t2* indica el tipo del slot correspondiente. La faceta type es una faceta obligatoria para cada slot, los posibles tipos que puede tener un slot son *integer*, *real*, *boolean*, *intlist*, *realist* y *array*. Además, en el caso de las listas, tanto de enteros como de reales, es posible indicar el máximo número de elementos que éstas admiten, tomando el valor por defecto cuando no se indica nada ni se inicializa la variable. Por su parte, en el caso de los vectores es obligatorio declarar un tamaño y especificar el tipo básico.

- La faceta default es opcional. *v1*, *v2* indican los valores por defecto que tomarán dichos slots en caso de que durante la instanciación no se indique valor alguno. Se debe ser especialmente cuidadoso en lo referente al proceso de instanciación cuando la clase tiene listas implicadas, debido al tamaño máximo de las mismas. El hecho de que se indique un valor por defecto supone que cuando no se indica el tamaño máximo de la lista, éste quede establecido en el tamaño del valor por defecto. Por otra parte, si durante la instanciación a un slot de tipo lista se le asigna algún valor, el tamaño máximo de la lista quedará establecido al número de elementos con los que se instancia el slot, independientemente de lo que se indique el la declaración de la clase. Esto es debido a que la declaración de la instancia prevalezca ante la declaración de la clase.

El siguiente código declara una clase que contiene slots de todos los tipos disponibles. Algunos de los slots se inicializan a valores por defecto.

```
(defclass tipos
   (slot int (type integer) (default 55))
   (slot sreal (type real))
   (slot bool (type boolean) (default true))
   (slot il (type intlist)) ;Tamaño máximo de lista predeterminado
   (slot ir (type realist) (default (1.0 2.0))) ;Tamaño máximo de lista 2
   (slot il7 (type intlist[7])) ;Tamaño máximo de lista 7
   (slot lini (type intlist[7]) (default (1 2 3))) ;Tamaño máximo de lista 7
   (slot v (type array[3] of integer) (default (0 0 0))) ; Vector de 3 elementos
)
```

9. Instancias

Las instancias forman la memoria de trabajo un sistema basado en reglas. Las instancias pueden ser de clases predefinidas ("integer", "real", "boolean", "intlist", "realist" o "array") o definidas por el usuario. Su declaración se ajusta a la siguiente sintaxis:

```
(definstances id_instancias
   (ibase_1 of tipo_i valor_1)
   (ibase_2 of tipo_j valor_2)
    ...
   (iclase_1 of clase_i
        (atributo_i1 valor_i1)
        (atributo_i2 valor_i2)
        ...
   )
   (iclase_2 of clase_j
        (atributo_j1 valor_j1)
        (atributo_j2 valor_j2)
        ...
    )
    ...
)
```

Como se puede observar, existen ciertas diferencias entre las instancias de los tipos básicos soportados por el lenguaje y las instancias de clases definidas por el usuario.

- *id_instancias* es el nombre dado a la declaración de instancias.
- *ibase_1*, *ibase_2*, *iclase_1*, *iclase_2* son identificadores de instancias.
- *tipo_i*, *tipo_j* son los tipos de las instancias *ibase_1* e *ibase_2* respectivamente. *tipo_i*, *tipo_j* son tipos básicos de Arlips. Por tanto es posible la declaración de instancias de tipo integer, real, boolean, intlist, realist o array de cualquier tipo simple. El tamaño máximo de la lista en caso de instancias de este tipo vendrá dado por la declaración explicita que realice el usuario, como criterio más importante. Si no se realizase una declaración de tamaño máximo de lista de forma explicita, pero la instancia se inicializase, el tamaño máximo sería el de la lista de inicialización de la instancia. Si tampoco se inicializase, el tamaño máximo de la lista vendría dado por el valor por defecto.
- *clase_i*, *clase_j* son los respectivos tipos de las instancias *iclase_1* e *iclase_2*. En este caso se trata de clases previamente definidas por el usuario.

- *valor_1*, *valor_2* son los valores que toman respectivamente las instancias *ibase_1* e *ibase_2* en su declaración. Aunque es posible la omisión de este campo, esto supone la no inicialización de la instancia, y por tanto el riesgo de que se produzcan comportamientos no deseados.

- Los pares (*atributo_i1 valor_i1*), (*atributo_i2 valor_i2*), (*atributo_j1i valor_j1*), (*atributo_j2 valor_j2*) suponen la inicialización de los slots de clase de nombre *atributo_i1*, *atributo_i2* de la clase *clase_i*, y *atributo_j1*, *atributo_j2* de la clase *clase_j* a los valores *valor_i1*, *valor_i2*, *valor_j1* y *valor_j1* respectivamente. El valor de inicialización debe ser compatible con el tipo del slot, que fue declarado en la declaración de la clase. De forma equiparable a lo que sucede con instancias de tipos básicos, no es obligatorio inicializar todos los slots de una instancia de clase. Sin embargo, nuevamente se recomienda la inicialización de todos los slots, bien en la declaración de la clase, bien en la declaración de la instancia, para evitar comportamientos inesperados.

El siguiente código declara una instancia de cada uno de los tipos vistos anteriormente. La declaración se ha dividido en dos partes, una con las instancias de tipos básicos del lenguaje, y otra con una instancia de la clase *tipos* mostrada en la declaración de clases.

```
(definstances tbasicos
   (iint of integer 2)
   (ireal of real 4.5)
   (ilistaint of intlist (1 2 3))
   (ilistareal of realist)
   (j of array[5] of integer (0 1 2 3 4))
 )

(definstances clases
   (iclase of tipos
      (int 23) ;Redefinición del valor por defecto.
      (sreal 3.0)  ;Se deja en valor por defecto del slot bool
      (il (8 8 8))
      (il7 (0 1 2)) ; Tamaño actual de la lista = 3. Tamaño máximo = 7
   )
)
```

10. Variables globales

Las variables globales son identificadores cuyo valor asignado puede cambiar durante la ejecución del sistema de producción. Sin embargo, a diferencia de las instancias, el cambio de valor de una variable global no provoca la ejecución de la fase de *pattern matching*. Por lo tanto. la modificación del valor de una variable global no posibilita por si sola la activación ni desactivación de reglas.

La declaración de variables globales se realiza de la siguiente manera:

```
(defvar
   (var_i (type tipo_i) (default valor_i))
   (var_j (typetipo_j) (default valor_j))
   ...
)
```

donde

- *var_i*, *var_j* son identificadores de variables.
- *tipo_i*, *tipo_j* son los tipos de las respectivas variables. Los posibles tipos admitidos son *integer*, *real*, *boolean*, *intlist*, *realist* y *array*. También es posible declarar variables de clases definidas por el usuario, y en este caso, el valor por defecto será el indicado en la declaración de la clase (no puede volver a indicarse en la declaración de la instancia). En el caso de variables de tipo lista, el tamaño máximo de la lista variable será el del valor por defecto para listas en las que no se especifica tamaño ni se inicializan.
- La faceta *default* es opcional y sirve para inicializar las variables a los valores *valor_i*, *valor_j* en la inicialización del sistema de producción.

Las variables globales pueden ser usadas en las partes derechas de las reglas, así como en los parámetros de funciones tanto internas como externas. Las variables globales también pueden ser usadas en la parte izquierda de las reglas, en las condiciones de activación de las reglas. Pero es necesario recordar que lo que diferencia a las variables globales de cualquier instancia es que no influirán en la activación del pattern matching, es decir, su cambio de valor no provocará la activación o desactivación de reglas.

El siguiente código realiza una declaración de cada uno de los tipos de variables globales soportados. Nótese que la faceta *default* es opcional:

```
(defvar
   {Declaración de vble global de tipo integer iniciándola a 0}
   (vint  (type integer) (default 0))

   {Declaración de vble global de tipo real. Se inicializará con 6 dígitos
     decimales en el código generado: 0.123457}
   (vreal (type real) (default 0.123456789))

   {Declaración de vble de tipo boolean inicializándola true}
   (vbool (type boolean) (default true))

   {Declaración de vble de tipo intlist. Se inicializa como una lista
```

```
  vacía que admitirá hasta 256 elementos.}
(vli1 (type intlist))

{Declaración de vble de tipo intlist. Inicializada
 con la lista (1 2). tamaño máximo=2.}
 (vli2 (type intlist) (default (1 2)))

{Declaración de vble de tipo realist. Tamaño máximo=7.
  Se inicializa con (1.0 2.0)}
 (vlr4 (type realist[7]) (default (1 2)))

{Declaración de vble de tipo array de 3 elementos reales}
 (vi1 (type array[3] of real) (default (1 2 3)))

{Declaración de vble de tipo class (previamente declarada).
  No es posible cambiar los valores pode defecto, que heredará de
  la clase class.}
  (vclass (type class))
```

11. Funciones

En Arlips se definen tres tipos de funciones:

Funciones predefinidas para el manejo de listas, salida/entrada,...

Funciones definidas por el usuario que permiten agrupar acciones Arlips, dotando a los programas de mayor claridad y grado de abstracción.

Funciones externas escritas en el lenguaje C que pueden ser invocadas desde los programas Arlips.

En la siguiente sección se explica el funcionamiento de estos tipos de funciones.

11.1. Otras funciones predefinidas

En otras secciones de este manual se han presentado varias funciones predefinidas (como las disponibles para la manipulación de listas). En esta sección se presentan algunas nuevas.

11.1.1. print

print permite imprimir información por pantalla. *print* es una sentencia que no devuelve valor alguno, por tanto no puede ser usada en la parte izquierda de una regla. La utilización de *print* es la que sigue:

```
(print(exp1, exp2,...))
```

donde *exp1*, *exp2* puede ser cualquier tipo básico del lenguaje (*integer*, *real*, listas *intlist* y *realist*, *boolean*, que será impreso como un int de C, *array* de un tipo simple o una cadena de tamaño acotado a 255 elementos[6]. Dado que las cadenas se copian literalmente a C, cualquier secuencia que sea una secuencia especial de caracteres en C, tendrá el mismo significado que tiene en C. Por ejemplo, la cadena *"\n"* significará el retorno de carro.

11.1.2. halt

La sentencia Halt permite parar la ejecución del sistema de producción en cualquier momento. Su utilización no requiere de ningún tipo de parámetro, y puede aparecer tanto en el consecuente de las reglas, como en el interior de las funciones declaradas en Arlips.

11.2. Funciones declaradas por el usuario

En Arlips es posible agrupar acciones en funciones para facilitar la construcción de sistemas complejos. Las funciones aceptan parámetros por valor y por referencia, y pueden devolver valores de tipo simple (*integer*, *real* o *boolean*, pero no listas) o no devolver nada. Las funciones pueden ser usadas como cualquier otra acción, o como una expresión en el caso de que devuelvan algún valor. Las funciones se declaran de la siguiente forma:

```
(deffunction f [(param1[*]tipo1 , param2 [*]tipo2,...)] [:tipo_retorno]
   (
     (accion1)
     (accion2)
      ...
     [return(expresion)]
   )
)
```

donde

- *f* es el identificador de la función
- En el caso de que la función tenga parámetros, estos deben aparecer en una lista encerrada entre paréntesis cuyos elementos son pares identificador tipo, y en la que cada par se separa por una coma del siguiente. Así *param1*, y *param2* son identificadores de parámetros. Mientras que *tipo1* y *tipo2* son los tipos de los respectivos parámetros. En el caso de tratarse de una función que no requiera pase de parámetros, los paréntesis no deben escribirse. Los parámetros pueden ser pasados

[6] Una cadena es una secuencia de caracteres comprendida entre unas comillas dobles que indican el inicio de la cadena, y las siguientes comillas dobles que indican el final de la misma

por valor o por referencia. Cuando los parámetros son pasados por referencia, se indica poniendo un * delante del tipo del parámetro. Los parámetros pueden ser de tipo simple o definido por el usuario.

- *:tipo_retorno* indica el tipo del dato devuelto por la función en los casos en los que devuelva alguno. El tipo de retorno puede ser *integer*, *real* o *boolean*. En las funciones que no devuelven valor alguno, esto se indica mediante la ausencia de este campo.

- *accion1*, *accion2*, ... son las acciones que debe realizar la función.

- Las funciones que devuelvan algún tipo simple, lo realizan mediante la acción especial *return(expresion)*, donde *expresion* es un valor o expresión compatible con el tipo devuelto por la función

Cuando se usan funciones, deben tenerse en cuenta ciertos detalles. Las acciones de las funciones solo pueden hacer referencia a instancias, a variables globales y a parámetros. No es posible usar variables patrón universalmente cuantificadas ni tan siquiera como un parámetro, pues este tipo de variables no hacen referencia a una única instancia.

La siguiente función calcula el cuadrado de un número entero pasado por valor, y lo devuelve.

```
(deffunction cuadrado (dato integer):integer
   (
     (return (dato*dato))
   )
 )
```

El lenguaje no impide que las funciones que devuelven algún valor puedan ser usadas en la parte izquierda de las reglas, aunque se debe tener cuidado con los efectos colaterales de estas funciones, ya que solo las modificaciones de instancias realizadas en el lado derecho de las reglas provocarán la activación o desactivación de reglas.

11.3. Funciones externas

Arlips soporta el uso de funciones externas que pueden ser usadas como acciones o como parte de expresiones. De esta forma se permite integrar sistemas basados en reglas Arlips con otros lenguajes de programación como C. La declaración de una función externa se realiza de la siguiente forma:

```
(extern funcion (tipos_parametros) :tipo_retorno)
```

donde

- *funcion* es el identificador externo de la función.
- *tipos_parametros* es una lista ordenada de los tipos de los parámetros en el lenguaje C tal y como serían escritos en la declaración de la función (por tanto *int*, *float*,...).
- *:tipo_retorno* indica el tipo devuelto por la función acorde a los tipos simples admitidos por Arlips (*integer*, *real* o *boolean*). Si la función no devuelve valor alguno, no debe escribirse este campo.

Es deseable aunque no obligatorio que el compilador de Arlips conozca dónde se encuentran definidas las funciones externas, para así poder generar correctamente el fichero *Makefile*. La directiva *extfile* que se ha explicado anteriormente permite esto. Gracias a esto se podrá indicar en el fichero *Makefile* al compilador de C que debe linkar el SBR creado con un fichero objeto que contiene el código binario de las funciones externas. Por ejemplo, si se desease usar una función que calculase la operación módulo-i de un número dado, cuyo código binario se encontrase en el fichero *mod.o*, se debería escribir el siguiente código:

```
#extfile "mod.o"
(extern mod(int a, int i):integer)
```

La primera de estas líneas indica al compilador Arlips que debe preparar el *Makefile* de forma que el código propio del SBR sea enlazado con el que se encuentra en el fichero binario *mod.o*. En la segunda linea se le indica al compilador Arlips que *mod* es una función externa que admite dos parámetros de tipo *int* [7], y que a su vez devuelve un valor entero. Con ello las reglas ya pueden usar la función *mod*.

12. Estrategia de control

Arlips permite definir la estrategia de control que seguirá la ejecución del sistema de producción. La estrategia de control se encarga de decidir qué instancia de regla se elegirá para ser disparada de entre las que están activas (las que forman el conjunto conflicto). Independientemente de la estrategia de control elegida, siempre se elegirá la regla activa más prioritaria. Solo cuando se debe elegir entre más de una regla con la misma prioridad se tendrá en cuenta la estrategia de control elegida.

El comando utilizado en Arlips para definir la estrategia de control que utilizará el sistema de producción es *set_strategy*. Su utilización es muy simple y sigue la siguiente sintaxis:

[7] Arlips no realizará comprobación alguna del número ni tipo de los parámetros, sin embago esto permitirá declarar la función en el código C generado

```
( set_strategy  estrategia )
```

Aquí estrategia puede ser una de las siguientes palabras reservadas: *depth* para denotar una estrategia en profundidad; o *width* para utilizar una estrategia en anchura.

La estrategia por defecto (la estrategia que se utiliza si no se especifica nada en el sistema de producción) es la estrategia en profundidad.

La estrategia en profundidad hará que de entre las reglas activas con la mayor prioridad se elija la que se activó más recientemente (la última en incorporarse a la agenda). La estrategia en anchura, hará que se escoja la instancia más antigua (la primera que se activó, incorporándose a la agenda).

13. Reglas

Las reglas son la parte activa de un sistema de producción. De hecho todo sistema de producción debe contener como mínimo una regla definida. Una regla se compone de dos partes: la condición de activación, antecedente o parte izquierda de la regla (LHS: "Left-Hand Side"), y la lista de acciones, consecuente o parte derecha de la regla (RHS: "Rigth-Hand Side"). El antecedente está separado del consecuente por el símbolo "=>". La sintaxis para definir una regla es la siguiente:

```
(defrule regla [prioridad]
     (condicion1)
     (condicion2)
     ...
  =>
     (accion1)
     (accion2)
     ...
```

donde

- *regla* es un nombre identificador de la regla que no debe coincidir con ninguna palabra reservada ni con el identificador de alguna declaración realizada por el usuario. Su función es meramente descriptiva.

- *prioridad* es un atributo opcional que sirve para determinar la prioridad de la regla. La prioridad determina el orden de selección de las reglas de la agenda (reglas cuyo antecedente se satisface con el contenido actual de la memoria de trabajo). Los valores admisibles para establecer la prioridad son los enteros comprendidos entre *-10* y *10*, siendo 10 la prioridad más alta. Arlips permite que la prioridad de la reglas se defina de forma dinámica (en tiempo de ejecución del sistema de producción). Para ello se puede usar una instancia o variable global

(de tipo entero) cuyo valor en tiempo de ejecución determinará el valor de la prioridad. En caso de usar prioridad dinámica, es responsabilidad del usuario garantizar que la prioridad no sale de su rango admisible.

- Las condiciones de una regla forman la parte izquierda de la misma, y se explican más detalladamente en el siguiente punto.
- Las acciones de la regla forman la parte derecha de las misma, y se explican tras el punto de las condiciones.

A continuación se detalla cada una de las partes.

13.1. Parte izquierda de las reglas

La parte izquierda de una regla, o condición de activación está formada por un conjunto de condiciones encerradas cada una de ellas entre paréntesis, y relacionadas implícitamente mediante el operador lógico AND. Las condiciones de activación son expresiones lógicas que se evalúan a cierto o falso. Las expresiones válidas y la evaluación que de ellas se realiza nuevamente está marcada por el lenguajes al que se traduce el código Arlips. Las condiciones de activación pueden ser:

- Una expresión lógica o numérica suelta, evaluable al estilo de C.
- Comparaciones entre expresiones numéricas.

```
(a op b)
```

  donde *op* puede ser

  - Igualdad =
  - Desigualdad <>
  - Mayor >
  - Mayor o igual >=
  - Menor <
  - Menor o igual <=

  Por su parte, tanto *a* como *b* pueden ser constantes, instancias de tipo integer o real, valores numéricos o expresiones compuestas por cualquiera de los elementos descritos anteriormente, relacionados por operadores aritméticos de suma, resta, multiplicación y división.

- Expresiones lógicas

- Se asume siempre un *and* implícito entre condiciones, a no ser que se especifique lo contrario.
- Es posible el uso del operador *or* binario de la siguiente forma:

```
(OR
   (a)
   (b)
)
```

  donde *a*, *b* pueden ser nuevamente una condición o secuencia de condiciones relacionadas por el operador implícito *and*[8].
- *not(a)* donde a es una condición en la que no ocurre ninguna variable patrón cuantificada existencialmente[9].

De la lista anterior, tal vez el elemento más sorprendente sea el primero. Es posible escribir como condición de activación un sencillo identificador de instancia de un elemento de tipo *integer*, *real* o *boolean*. En la evaluación, dicho identificador será evaluado por su valor, considerándose como falso cualquier valor igual a *0* (o a *0.0*), y cierto *true* cualquier otro valor.

Para formar condiciones se pueden usar instancias de la memoria de trabajo, variables patrón, constantes, o funciones, tanto propias del lenguaje como definidas por el usuario, que no modifiquen la memoria de trabajo. Arlips tampoco impide el uso de funciones que modifiquen la memoria de trabajo o de variables, aunque esto no es recomendable ya que los efectos colaterales de las funciones no activarán ni desactivarán reglas.

El algoritmo de pattern matching empleado en Arlips, al igual que el de otros sistemas de producción, comienza evaluando las condiciones de cada regla tras la inicialización de la memoria de trabajo. Tras la ejecución de cada regla, se vuelven a evaluar las condiciones de cada regla para determinar de nuevo el conjunto conflicto (reglas cuyas condiciones del antecedente se evalúan a cierto con el contenido en ese momento de la memoria de trabajo). Si la evaluación de todas las condiciones de una regla resulta cierta, entonces la regla se anota en la agenda, junto con las instanciaciones de las variables patrón que han producido esta activación y la prioridad de la misma.

[8] En la versión actual de Arlips, no soporta el anidamiento de operadores *or*.

[9] Nótese que esto no limita la potencia del lenguaje. Expresar que no existe un elemento que cumple una determinada condición es equivalente a expresar que todos los elementos no cumplen una condición, o que cumplen la condición inversa. Por tanto es posible el uso de un cuantificador universal. Sin embargo, semánticamente haber permitido la negación de una variable patrón cuantificada existencialmente planteaba una inconsistencia, pues dichas variables siempre se instancian a un determinado elemento en una determinada instanciación de la regla pero, ¿a cuál de todos los candidatos se debía instanciar la variable patrón cuando la condición expresa que ninguno de ellos debe satisfacer *a*?

De todas las reglas anotadas en la agenda para su ejecución, pasa a ejecución la que tenga la prioridad más alta. Cuando varias reglas tienen la misma prioridad se considera la estrategia de control del sistema de producción: *profundidad*, en la que se ejecutará primero la regla que se ha introducido en la agenda más recientemente; o o *anchura*, en cuyo caso se ejecutará primero la regla más 'antiguas' (la primera que fue introducida en la agenda).

13.2. Parte derecha de las reglas

La parte derecha de la regla o RHS se corresponde con la lista de acciones de la regla. Las acciones pueden implicar a constantes, variables globales, instancias, funciones proporcionadas por Arlips, externas, o definidas por el usuario, así como variables patrón instanciadas existencialmente (ya que éstas hacen referencia a instancias).[10]

Las principales acciones con las que cuenta el lenguaje Arlips son la asignación y la llamada a una función. La asignación se realiza mediante el operador "*:=*"(similar al lenguaje Pascal):

```
(a:=b)
```

que otorga a la instancia básica, variable global básica, o slot de una variable patrón, instancia o variable de clases definidas por el usuario el valor de la expresión *b*. *b* puede ser cualquier expresión compatible con el tipo de *a*, así puede ser un valor numérico, una constante, una instancia o variable de tipo básico, una función, un slot de una variable patrón, instancia o variable de clases definidas por el usuario, o cualquier combinación de las anteriores relacionadas mediante operadores aritméticos, siempre que se respete la compatibilidad de tipos.

En la parte de las acciones es posible llamar a todo tipo de funciones. Las funciones que devuelvan algún valor deberán estar asignadas a algún elemento compatible con dicho valor. Las que no lo hagan pueden ser llamadas directamente. Sea el ejemplo:

```
(f(a,b))
(c:=g)
```

donde la función *f* admite dos parámetros y no devuelve valor alguno. Por su parte, la función *g* no requiere de parámetros (y tampoco se usan los paréntesis de la lista de parámetros), y devuelve un valor compatible con el tipo de *c*, que es asignado a *c*.

También es posible la modificación de listas, la impresión por pantalla, la modificación de la prioridad o de la estrategia de control, o la parada del sistema, tal y como se explica en las respectivas secciones.

[10] No se permite el uso de variables globales cuantificadas universalmente.

Aquellas acciones que produzcan cambios en la memoria de trabajo, y por tanto que modifiquen el valor de alguna instancia, activarán la fase de pattern matching, reevaluando las condiciones que se vean afectadas por dichos cambios.

Un ejemplo de declaración de una regla sería:

```
(defconst (LIMITE_DER 10))

(defclass posicion
     (slot posx (type INTEGER))
     (slot posy (type INTEGER))
  )

(defpattern (x  POSICION))

(defrule mueve_der [5]
  ( ?x.posx<LIMITE_DER )
  ( NOT (casilla_ocupada ( ?x.posx+1, ?x.posy) ))
 =>
  (?x.posx :=?x.posx+1)
)
```

14. Variables patrón

Las variables patrón son un tipo especial de variables que pueden instanciarse dinámicamente en la parte izquierda de las reglas al valor de las instancias que cumplan las condiciones en las que aparece la variable patrón. Las variables patrón producen reevaluación de la condición de activación de una regla ante el cambio de valor de cualquier instancia de la clase a la que referencia dicha variable patrón.

Las variables patrón deben declararse de la siguiente forma:

```
(defpattern
  (vp1 clase_i)
  (vp2 clase_j)
   ...
)
```

- *vp1*, *vp2* son identificadores de variables patrón
- *clase_i*, *clase_j* son las clases de las variables patrón *vp1* y *vp2* respectivamente.

Como se puede observar, solo es posible la declaración de variables patrón para clases definidas por el usuario. Las variables patrón que se declaren para

una determinada clase, solo evaluarán elementos de dicha clase, pero no de clases padre o hijo.[11]

Las variables patrón se instancian en el lado izquierdo de las reglas. Es posible cuantificar universal o existencialmente las variables patrón. La cuantificación existencial de una variable patrón se representa mediante el símbolo *?* precediendo al identificador de la variables, y la cuantificación universal se representa por el símbolo *@*. La cuantificación existencial provocará que para que la regla se dispare, la variable patrón deba instanciarse a alguna de las instancias de la misma clase de la variable, y que cumpla la restricción establecida a la variable patrón. Por su parte, para que una regla con una variable patrón cuantificada universalmente pueda dispararse, se exige que todas las instancias de la misma clase que la variable patrón cumplan con la condición impuesta sobre la variable patrón. Las variables patrón cuantificadas existencialmente pueden ser usadas en la parte derecha de la regla que la cuantifica. Por ejemplo, una asignación de un nuevo valor a algún slot de la variable patrón significará el cambio de valor de dicho slot en la instancia a la que se instanció la variable patrón en la parte izquierda de la regla. Las variables patrón cuantificadas universalmente no pueden usarse en la parte derecha de una regla. Recuerde no combinar nunca un cuantificador existencial y uno universal sobre una única variable patrón en distintas condiciones de una misma regla.

Por ejemplo, dada la clase:

```
(defclass sonar
   (slot id (type integer))
   (slot dist (type integer))
   (slot critico (type boolean) (default false))
 )
```

Se puede declarar la variable patrón *vp* de tipo *sonar* de la siguiente manera:

```
(defpattern
   (vp sonar)
 )
```

Supongamos que una regla requiere para su activación que todos los sonares tengan en su campo *dist* un valor superior a 50 unidades, eso, dentro de una regla se puede expresar como:

```
   ...
  (@vp>50)
   ...
```

[11] No es posible definir una variable patrón para un slot. Solo se permite definir variables patrón de clases.

```
 =>
   ...
```

Supóngase que existiera una regla que marcase como crítico a cualquier sonar cuyo valor en el campo *dist* sea menor de 50. Codificar que cuando el sonar devuelva una distancia superior a 50 deje de ser crítico se podría realizar de la siguiente forma:

```
(defrule no_critico
   (?vp.critico = true)
   (?vp.dist > 50)
 =>
   (?vp.critico:=false)
)
```

La variable patrón se instancia una vez a alguno de los sonares, y una vez instanciada, cada aparición de la misma en la regla hace referencia a la misma instanciación, es por ello que sea el mismo sonar el que debe ser crítico, que su distancia sea mayor que 50, y que tras la ejecución de esta regla dejará de ser crítico. Si se desean usar dos instanciaciones distintas, se deberán declarar y usar dos variables patrón distintas, y además comprobar que las dos variables patrón no se han instanciado a la misma instancia (exigiendo por ejemplo, que el slot id sea distinto para cada una de las instanciaciones).

15. Depuración de un sistema de producción

Arlips ofrece una serie de características que facilitan la depuración de los sistemas de producción, como por ejemplo la generación de un registro de los eventos de la agenda del sistema, o la ejecución por pasos o parcial del sistema de producción generado.

15.1. Ejecución del sistema basado en reglas por pasos o con límite

Siempre y cuando no haya que empotrar el sistema de producción Arlips con software externo, es decir, siempre que no se importe un *kdm* y además no se escoja ni *-nomain* ni *-artis* en las opciones de compilación, Arlips generará código capaz de realizar ejecuciones parciales y/o por pasos del sistema de producción. Al ejecutar el sistema de producción podrá combinar las opciones *-limit* y *-steps* para ello.

- *-limit l* establece el límite de pasos de ejecución que el sistema de producción podrá realizar a un máximo de *l*. Una vez alcanzada esa cifra el sistema finalizará con independencia de su estado.

- *-steps s* ejecuta hasta *s* pasos del sistema de producción. Si el sistema basado en reglas no ha finalizado tras la ejecución de los *s* pasos, el sistema se detiene esperando la interacción con el usuario.

    - Cualquier cadena que comience por *q* o *Q* será interpretada como una orden quit, es decir, finalizará el sistema basado en reglas inmediatamente.
    - Cualquier número positivo *n* sobreescribirá el valor de *s*, estableciendo la siguiente parada del sistema de producción a la ejecución de *n* nuevos pasos.
    - *0* implicará la desactivación de la opción *-steps*. El sistema continuará su ejecución sin interrupción hasta su finalización o hasta que se alcance el límite de pasos si éste se hubiese establecido.
    - Cualquier otra cadena que no satisfaga alguna de las condiciones anteriores, incluyendo en estas los números negativos, supondrá la continuidad del sistema de producción otros *n* pasos más, siendo *n* el número de pasos ejecutados desde la parada anterior hasta ésta, es decir, el más reciente de los valores de *s*.

- Cualquier otro argumento producirá la impresión de una ayuda por pantalla.

15.2. La acción *facts*

La acción *facts* es una orden que puede ser escrita en la parte derecha de una regla y que solo tiene efectos cuando el sistema de producción ha sido generado con la opción *-log*. *facts* supone el volcado de toda la memoria de trabajo, incluyendo las variables globales, al registro de ejecución del sistema. En caso de que el sistema de producción no haya sido generado con la opción *-log*, *facts* será ignorada. En la sección 15.3 se puede encontrar una descripción de la estructura del fichero log

15.3. Opción de compilación *-log*

-log genera código adicional que permite registrar determinados eventos del sistema en un fichero con formato xml. En el cuadro 1 se puede observar el log de un sistema de producción. Las etiquetas utilizadas y sus posibles valores son las siguientes:

- $< arlips\_log >$ etiqueta que abarca todo el registro de una ejecución completa del sistema de producción especificado por el atributo *RBS*

- $< reset >$ etiqueta que indica la ejecución de la función reset del sistema de producción. Su contenido es la lista de acciones que produce.

- $< fired >$ registra la ejecución de una instanciación de regla presente en la agenda, con la prioridad indicada. Contiene la lista de acciones que provoca dicha ejecución.

- $< actions >$ etiqueta que engloba una lista de acciones. Se puede encontrar encerrada por la etiqueta *reset* o por *fired*. Las posibles acciones que se registran son la activación o desactivación de una instanciación de una regla, o la ejecución de la orden *facts* que provoca el registro de toda la memoria de trabajo.

- $< activation >$ etiqueta que indica la activación de una instanciación de la regla contenida en su interior junto con la prioridad indicada.

- $< deactivation >$ registra la desactivación de una instancia de regla de la agenda.

- $< rule >$ etiqueta que encierra el nombre de una regla. Aparece en el interior de las etiquetas de activación, desactivación y disparo de una regla.

- $< priority >$ indica la prioridad con la que se produce la activación o disparo de una instancia de regla.

- $< end >$ registra el motivo del fin de ejecución del sistema de producción. Los posibles motivos contemplados son:

    - *empty*: El sistema basado en reglas acabó porque la agenda se vació y no hubo más reglas que disparar.
    - *halt*: La última regla ejecutada realizó una llamada explícita a la orden *halt*, finalizando la ejecución del sistema.
    - *limit*: Se realizó una ejecución del sistema basado en reglas usando la opción *-limit*, el sistema finalizó por haber alcanzado el límite de pasos de ejecución impuesto por el usuario.
    - *quit*: El sistema de producción se ejecutó con la opción *-steps* y en una de las paradas de paso el usuario escogió la opción *quit*, finalizando el sistema basado en reglas.

- $< working\_memory >$ es una etiqueta contenida en la lista de acciones que representa la ejecución de una acción *facts*. Esta etiqueta encierra el estado de la memoria de trabajo en el momento en que se ejecutó facts. En su interior se encuentran las distintas clases existentes en el sistema de producción.

- $< class >$ etiqueta cuyo significado es la representación de la clase indicada por su atributo *id*. Su aparición ocurre en el interior de etiquetas *working_memory* y su misión es contener todas las instancias y

variables existentes de la clase indicada por *id*. Cuando las instancias o variables no son de un tipo definido por el usuario el valor del atributo *id* será *"system"*.

- $< instances >$ etiqueta usada para agrupar el conjunto de instancias de una determinada clase. Su contenido son etiquetas que representan el nombre de una instancia y éstas a su vez contendrán los nombres de los slots de clase con sus valores.

- $< variables >$ etiqueta usada para agrupar el conjunto de variables de una determinada clase. Al igual que en el caso de las instancias, esta etiqueta encierra etiquetas que representan el nombre de una variable y éstas a su vez contendrán los nombres de los slots de la clase con sus valores.

16. Lista de palabras reservadas

addhead	addtail	and	array
boolean	default	defclass	defconst
deffunction	definstances	defpattern	defrule
defvar	depth	extern	false
head	import	include	integer
intlist	isa	kdm	not
of	or	pophead	poptail
print	real	realist	return
setstrategy	size	slot	tail
true	type	width	

```
<arlips_log RBS="ejemplo" timestamp="0.000000">
  <reset>
    <actions>
      <activation>
        <rule>es_mod2</rule>
        <priority>0</priority>
      </activation>
      <activation>
        <rule>es_mod3</rule>
        <priority>0</priority>
      </activation>
      <activation>
        <rule>menor10</rule>
        <priority>0</priority>
      </activation>
    </actions>
  </reset>
  <fired>
    <rule>menor10</rule>
    <priority>0</priority>
    <actions>
      <deactivation>
        <rule>menor10</rule>
      </deactivation>
      <deactivation>
        <rule>es_mod2</rule>
      </deactivation>
      <deactivation>
        <rule>es_mod3</rule>
      </deactivation>
      <activation>
        <rule>menor10</rule>
        <priority>0</priority>
      </activation>
    </actions>
  </fired>
  <end>halt</end>
</arlips_log>
```

Cuadro 1: Ejemplo de *log* de un sistema de producción Arlips