Programación basada en reglas

Franciso J. Martín Mateos
José Luis Ruiz Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Ampliación de Inteligencia Artificial, 2014-15

Índice

Sistemas de producción

- Un *sistema de producción* es un mecanismo computacional basado en *reglas de producción* de la forma: “Si se cumplen las *condiciones* entonces se ejecutan las *acciones*”
- El conjunto de las reglas de producción forma la *base de conocimiento* que describe como evoluciona un sistema
  - Las reglas de producción actúan sobre una *memoria de trabajo* o *base de datos* que describe el estado actual del sistema
  - Si la condición de una regla de producción se satisface entonces dicha regla está *activa*
  - El conjunto de reglas de producción activas en un instante concreto forma el *conjunto de conflicto* o *agenda*
  - La *estrategia de resolución* selecciona una regla del conjunto de conflicto para ser ejecutada o *disparada* modificando así la memoria de trabajo

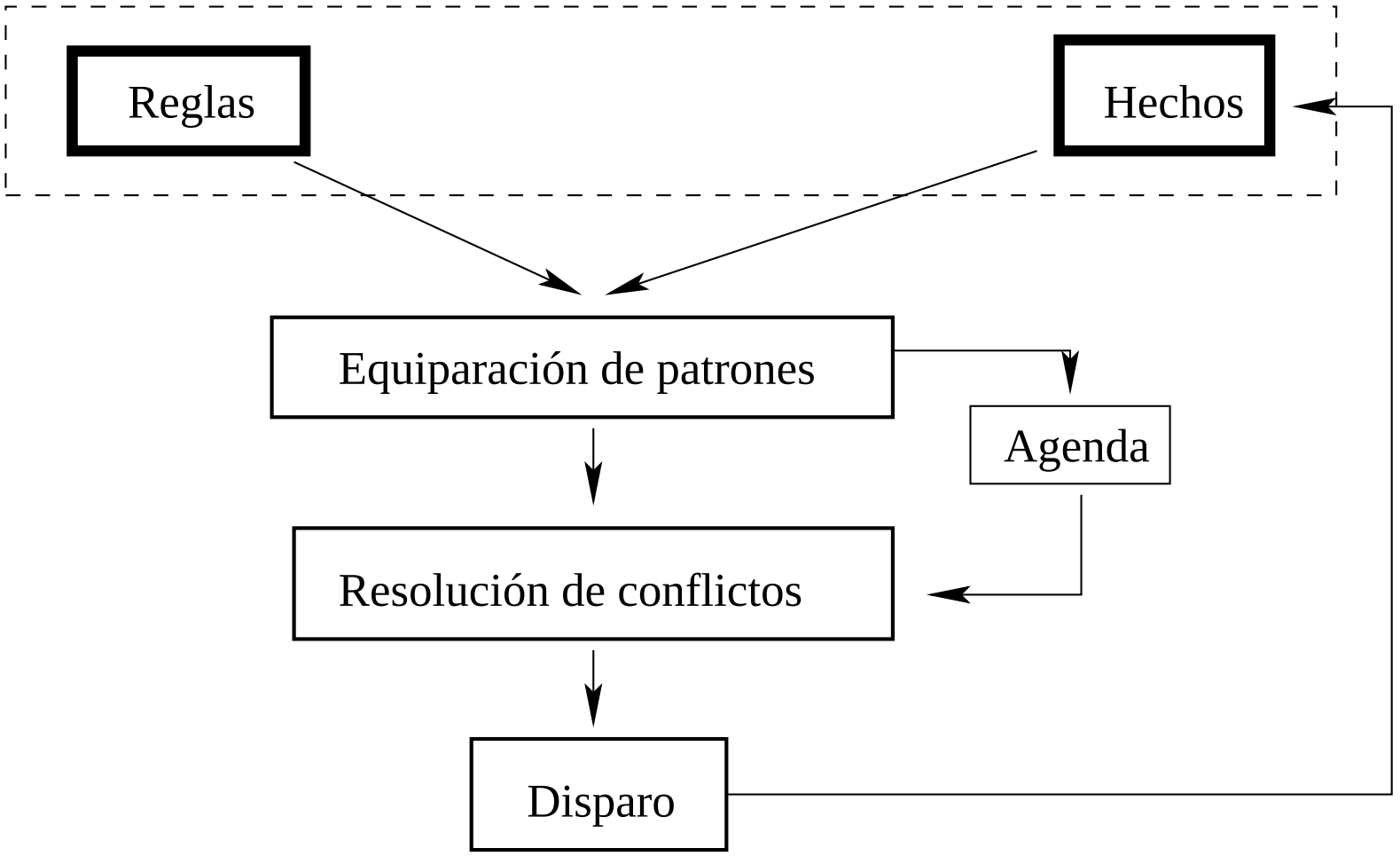
Sistemas de producción

- Componentes:
  - Base de hechos (*memoria de trabajo*). Elemento dinámico
  - Base de reglas (*base de conocimiento*). Elemento estático
  - Motor de inferencia (produce los cambios en la memoria de trabajo)
- Elementos adicionales:
  - *Algoritmo de equiparación de patrones*: Algoritmo para calcular **eficientemente** la agenda
  - *Estrategia de resolución de conflictos*: Proceso para decidir en cada momento qué regla de la agenda debe ser disparada

Resolución de conflictos

- Una activación sólo se produce una vez
- Estrategias más comunes:
  - Tratar la agenda como una pila
  - Tratar la agenda como una cola
  - Elección aleatoria
  - Regla más específica (número de condiciones)
  - Activación más reciente (en función de los hechos)
  - Regla menos utilizada
  - Mejor (pesos)

Ciclo de ejecución



Ejemplo de sistema de producción

- Reglas para identificar un animal
  - Si el animal tiene pelos entonces es mamífero
  - Si el animal produce leche entonces es mamífero
  - Si el animal es mamífero y tiene pezuñas entonces es ungulado
  - Si el animal es mamífero y rumia entonces es ungulado
  - Si el animal es ungulado y tiene el cuello largo entonces es una jirafa
  - Si el animal es ungulado y tiene rayas negras entonces es una cebra
- ¿Cómo identificamos un animal que tiene pelos, pezuñas y rayas negras?

Reglas de producción

- Modelo de regla:

  `<Condiciones> => <Acciones>`

- Condiciones
  - Existencia de cierta información
  - Ausencia de cierta información
  - Relaciones entre datos
- Acciones
  - Incluir nueva información
  - Eliminar información
  - Presentar información en pantalla

CLIPS

- CLIPS $\equiv$ C Language Integrated Production Systems
  - http://clipsrules.sourceforge.net/
- Lenguaje basado en reglas de producción.
- Desarrollado en el *Johnson Space Center* de la NASA.
- Relacionado con OPS5 y ART
- Características:
  - Conocimiento: Reglas, objetos y procedimental
  - Portabilidad: implementado en C
  - Integración y Extensibilidad: C, Java, FORTRAN, ADA
  - Documentación
  - Bajo coste: software libre

Hechos en CLIPS

- Estructura de un hecho simple
  `(<simbolo> <datos>*)`
- Ejemplos:
  - `(conjunto A 1 2 3)`
  - `(1 2 3 4)` no es un hecho válido
- Conjunto de hechos iniciales

```
(deffacts <nombre>
   <hecho>*)
```

- Ejemplo:

```
(deffacts datos-iniciales
  (conjunto A 1 2 3 4)
  (conjunto B 1 3 5))
```

Reglas en CLIPS

- Estructura de una regla (I):

```
(defrule <nombre>
  <condicion>*
  =>
  <accion>*)
```

- Las condiciones son patrones que se equiparan con los hechos de la memoria de trabajo
- Acción: Añadir hechos

```
(assert <hecho>*)
```

- Ejemplo:

```
(defrule mamifero-1
  (tiene-pelos)
  =>
  (assert (es-mamifero)))
```

Interacción con CLIPS

- Limpiar la base de conocimiento: `(clear)`
- Cargar el contenido de un archivo: `(load <archivo>)`
- Inicializar el sistema de producción: `(reset)`
- Visualizar la memoria de trabajo: `(facts)`
- Visualizar la agenda: `(agenda)`
- Ejecutar el sistema de producción: `(run)`
  - Ejecutar una regla en el sistema de producción: `(run 1)`
- Acceder a la ayuda del sistema: `(help)`
- Salir del sistema: `(exit)`

Ejemplo de sistema de producción en CLIPS

- Reglas

```
(defrule mamifero-1
  (tiene-pelos)
  =>
  (assert (es-mamifero)))

(defrule ungulado-1
  (es-mamifero)
  (tiene-pezugnas)
  =>
  (assert (es-ungulado)))

(defrule jirafa
  (es-ungulado)
  (tiene-cuello-largo)
  =>
  (assert (es-jirafa)))
```

```
(defrule mamifero-2
  (da-leche)
  =>
  (assert (es-mamifero)))

(defrule ungulado-2
  (es-mamifero)
  (rumia)
  =>
  (assert (es-ungulado)))

(defrule cebra
  (es-ungulado)
  (tiene-rayas-negras)
  =>
  (assert (es-cebra)))
```

- Conjunto de hechos iniciales

```
(deffacts hechos-iniciales
  (tiene-pelos)
  (tiene-pezugnas)
  (tiene-rayas-negras))
```

Seguimiento de la ejecución

- Visualizar las entradas y salidas de hechos

```
(watch facts)
```

- Visualizar las activaciones y desactivaciones de las reglas

```
(watch activations)
```

- Visualizar los disparos de las reglas

```
(watch rules)
```

- Desactivar el seguimiento

```
(unwatch facts)
(unwatch activations)
(unwatch rules)
```

Tabla de seguimiento

Hechos	E	Agenda	D
`f0 (initial-fact)`	0		
`f1 (tiene-pelos)`	0	`mamifero-1: f1`	1
`f2 (tiene-pezugnas)`	0		
`f3 (tiene-rayas-negras)`	0		
`f4 (es-mamifero)`	1	`ungulado-1: f4,f2`	2
`f5 (es-ungulado)`	2	`cebra: f5,f3`	3
`f6 (es-cebra)`			

Variables en CLIPS

- Variables simples: `?x, ?y`
  - Toman un valor simple (número, símbolo o cadena de texto)
- Variables múltiples: `$?x, $?y`
  - Toman como valor una secuencia de valores simples
- Variables mudas: Toman un valor que no es necesario recordar
  - Simple: `?`
  - Múltiple: `$?`

Reglas en CLIPS

- Estructura de una regla (II):

```
(defrule <nombre>
  <condicion>*
  =>
  <accion>*)
```

- Condiciones positivas y negativas

```
<condicion> := <patron> |
               (not <patron>) |
               <variable-simple> <- <patron>
```

  - Condiciones positivas: comprueban la presencia de un hecho
  - Condiciones negativas: comprueban la ausencia de un hecho

- Acción: eliminar hechos

```
(retract <identificador-hecho>*)

<identificador-hecho> := <variable-simple>
```

Unión de conjuntos

- Reglas

```
(defrule inicio
  =>
  (assert (union)))

(defrule union
  ?h <- (union $?u)
  (conjunto ? $? ?e $?)
  (not (union $? ?e $?))
  =>
  (retract ?h)
  (assert (union ?e $?u)))
```

- Conjunto de hechos iniciales

```
(deffacts datos-iniciales
  (conjunto A 1 2 3 4)
  (conjunto B 1 3 5))
```

Unión de conjuntos: tabla de seguimiento

Hechos	E	S	Agenda	D	S
`f0 (initial-fact)`	0		`inicio: f0`	1	
`f1 (conjunto A 1 2 3 4)`	0				
`f2 (conjunto B 1 3 5)`	0				
`f3 (union)`	1	2	`union: f3,f1(?e=4),`		2
			`union: f3,f1(?e=3),`		2
			`union: f3,f1(?e=2),`		2
			`union: f3,f1(?e=1),`		2
			`union: f3,f2(?e=5),`		2
			`union: f3,f2(?e=3),`		2
			`union: f3,f2(?e=1),`	2	
`f4 (union 1)`	2	3	`union: f4,f1(?e=4),`		3
			`union: f4,f1(?e=3),`		3
			`union: f4,f1(?e=2),`		3
			`union: f4,f2(?e=5),`		3
			`union: f4,f2(?e=3),`	3	
`f5 (union 3 1)`	3	4	`union: f5,f1(?e=4),`		4
			`union: f5,f1(?e=2),`		4
			`union: f5,f2(?e=5),`	4	
`f6 (union 5 3 1)`	4	5	`union: f6,f1(?e=4),`		5
			`union: f6,f1(?e=2),`	5	
`f7 (union 2 5 3 1)`	5	6	`union: f7,f1(?e=4),`		6
`f8 (union 4 2 5 3 1)`	6				

Intersección de conjuntos

- Reglas

```
(defrule inicio
  (conjunto ? $?s)
  (not (interseccion $?))
  =>
  (assert (interseccion $?s)))

(defrule interseccion
  ?h <- (interseccion $?i ?e $?f)
  (conjunto ?id $?)
  (not (conjunto ?id $? ?e $?))
  =>
  (retract ?h)
  (assert (interseccion $?i $?f)))
```

- Conjuntos de hechos iniciales

```
(deffacts datos-iniciales
  (conjunto A 1 2 3 4)
  (conjunto B 1 3 5))
```

Intersección de conjuntos: tabla de seguimiento

Hechos	E	S	Agenda	D	S
`f0 (initial-fact)`	0				
`f1 (conjunto A 1 2 3 4)`	0		`inicio: f1`		1
`f2 (conjunto B 1 3 5)`	0		`inicio: f2`	1	
`f3 (interseccion 1 3 5)`	1	2	`interseccion: f3,f1,`	2	
`f4 (interseccion 1 3)`	2				

Plantillas

- Estructura de una plantilla:

```
(deftemplate <nombre>
  <campo>*)

<campo> := (slot <nombre-campo>)
           (multislot <nombre-campo>)
```

  - Un campo simple `slot` tiene que almacenar exactamente un valor simple
  - Un campo múltiple `multislot` puede almacenar cualquier secuencia de valores

- Ejemplos:

```
(deftemplate conjunto
  (slot nombre)
  (multislot datos))

(conjunto (nombre A)
          (datos 1 2 3 4))

(conjunto (nombre B)
          (datos 1 3 5))
```

Unión de conjuntos con plantillas

- Plantillas

```
(deftemplate conjunto
  (slot nombre)
  (multislot datos))
```

- Reglas

```
(defrule inicio
  =>
  (assert (conjunto (nombre union) (datos))))

(defrule union
  ?h <- (conjunto (nombre union) (datos $?u))
  (conjunto (datos $? ?e $?))
  (not (conjunto (nombre union) (datos $? ?e $?)))
  =>
  (retract ?h)
  (assert (conjunto (nombre union) (datos ?e $?u))))
```

Intersección de conjuntos con plantillas

- Plantillas

```
(deftemplate conjunto
  (slot nombre)
  (multislot datos))
```

- Reglas

```
(defrule inicio
  (conjunto (datos $?s))
  (not (conjunto (nombre interseccion)))
  =>
  (assert (conjunto (nombre interseccion) (datos $?s))))

(defrule interseccion
  ?h <- (conjunto (nombre interseccion) (datos $?i ?e $?f))
  (conjunto (nombre ?id))
  (not (conjunto (nombre ?id) (datos $? ?e $?)))
  =>
  (retract ?h)
  (assert (conjunto (nombre interseccion) (datos $?i $?f))))
```

Restricciones sobre las variables

- Condiciones sobre las variables que se comprueban en el momento de analizar las condiciones de una regla
  - Negativas: `(dato ?x&~a)`
  - Disyuntivas: `(dato ?x&a|b)`
  - Conjuntivas: `(dato ?x&~a&~b)`
  - Igualdad: `(dato ?x&=<llamada-a-funcion>)`
  - Evaluables: `(dato ?x&:<llamada-a-predicado>)`
- Funciones y predicados en CLIPS
  - En la ayuda de CLIPS: `FUNCTION_SUMMARY`

Reglas en CLIPS

- Estructura de una regla (III):

```
(defrule <nombre>
  <condicion>*
  =>
  <accion>*)
```

- Condiciones explícitas: comprueban propiedades de las variables

```
<condicion> := <patron> |
               (not <patron>) |
               <variable-simple> <- <patron> |
               (test <llamada-a-predicado>)
```

- Acción: presentar información en pantalla

```
(printout t <dato>*)
```

Elementos repetidos en un tablero

- Plantillas

```
(deftemplate casilla
  (slot fila)
  (slot columna)
  (slot valor))
```

- Conjuntos de hechos iniciales

```
(deffacts datos-iniciales
  (casilla (fila 1) (columna 1) (valor 1))
  (casilla (fila 1) (columna 2) (valor 3))
  (casilla (fila 1) (columna 3) (valor 2))
  (casilla (fila 2) (columna 1) (valor 1))
  (casilla (fila 2) (columna 2) (valor 2))
  (casilla (fila 2) (columna 3) (valor 1))
  (casilla (fila 3) (columna 1) (valor 3))
  (casilla (fila 3) (columna 2) (valor 2))
  (casilla (fila 3) (columna 3) (valor 3)))
```

Elementos repetidos en un tablero

- Reglas

```
(defrule repetidos-en-la-misma-fila
  (casilla (fila ?f) (columna ?c1) (valor ?v))
  (casilla (fila ?f) (columna ?c2&~?c1) (valor ?v))
  =>
  (printout t "Repetidos en fila " ?f
              " y columnas " ?c1 " y " ?c2 t))

(defrule repetidos-en-la-misma-columna
  (casilla (fila ?f1) (columna ?c) (valor ?v))
  (casilla (fila ?f2&~?f1) (columna ?c) (valor ?v))
  =>
  (printout t "Repetidos en columna " ?c
              " y filas " ?f1 " y " ?f2 t))

(defrule repetidos-en-distintas-casillas
  (casilla (fila ?f1) (columna ?c1) (valor ?v))
  (casilla (fila ?f2) (columna ?c2) (valor ?v))
  (test (or (!= ?f1 ?f2) (!= ?c1 ?c2)))
  =>
  (printout t "Repetidos en casillas (" ?f1 "," ?c1
              ") y (" ?f2 "," ?c2 ")" t))
```

Ordenación de un vector

- Reglas

```
(defrule ordena
  ?f <- (vector $?b ?m1 ?m2&:(< ?m2 ?m1) $?e)
  =>
  (retract ?f)
  (assert (vector $?b ?m2 ?m1 $?e)))

(defrule resultado
  (vector $?x)
  (not (vector $?b ?m1 ?m2&:(< ?m2 ?m1) $?e))
  =>
  (printout t "El vector ordenado es " $?x t))
```

- Conjuntos de hechos iniciales

```
(deffacts datos-iniciales
  (vector 3 2 1 4))
```

Ordenación de un vector: tabla de seguimiento

Hechos	E	S	Agenda	D	S
`f0 (initial-fact)`	0				
`f1 (vector 3 2 1 4)`	0	1	`ordena: f1(?m1=2,?m2=1)`		1
			`ordena: f1(?m1=3,?m2=2)`	1	
`f2 (vector 2 3 1 4)`	1	2	`ordena: f2(?m1=3,?m2=1)`	2	
`f3 (vector 2 1 3 4)`	2	3	`ordena: f3(?m1=2,?m2=1)`	3	
`f4 (vector 1 2 3 4)`	3		`resultado: f0„f4`	4	

Clasificación de elementos

```
(deftemplate triangulo
  (slot nombre)
  (multislot lados))

(deffacts triangulos
  (triangulo (nombre A) (lados 3 4 5))
  (triangulo (nombre B) (lados 6 8 9))
  (triangulo (nombre C) (lados 6 8 10)))

(defrule inicio
  =>
  (assert (triangulos-rectangulos)))
```

Clasificación de elementos

```
(defrule almacena-triangulo-rectangulo
  ?h1 <- (triangulo (nombre ?n) (lados ?x ?y ?z))
  (test (= ?z (sqrt (+ (** ?x 2) (** ?y 2)))))
  ?h2 <- (triangulos-rectangulos $?a)
  =>
  (retract ?h1 ?h2)
  (assert (triangulos-rectangulos $?a ?n)))

(defrule elimina-triangulo-no-rectangulo
  ?h <- (triangulo (nombre ?n) (lados ?x ?y ?z))
  (test (!= ?z (sqrt (+ (** ?x 2) (** ?y 2)))))
  =>
  (retract ?h))

(defrule fin
  (not (triangulo))
  (triangulos-rectangulos $?a)
  =>
  (printout t "Lista de triangulos rectangulos: " $?a crlf))
```

Problemas de terminación

- Una misma regla se puede activar con distinto conjunto de hechos dando lugar a una iteración infinita en el proceso de deducción

```
(deftemplate rectangulo
  (slot nombre)
  (slot base)
  (slot altura))

(deffacts informacion-inicial
  (rectangulo (nombre A) (base 9) (altura 6))
  (rectangulo (nombre B) (base 7) (altura 5))
  (rectangulo (nombre C) (base 6) (altura 8))
  (rectangulo (nombre D) (base 2) (altura 5))
  (suma 0))

(defrule suma-areas-de-rectangulos
  (rectangulo (base ?base) (altura ?altura))
  ?suma <- (suma ?total)
  =>
  (retract ?suma)
  (assert (suma (+ ?total (* ?base ?altura)))))
```

Problemas de terminación

```
(defrule areas
  (rectangulo (nombre ?n) (base ?b) (altura ?h))
  =>
  (assert (area-a-sumar ?n (* ?b ?h))))

(defrule suma-areas-de-rectangulos
  ?nueva-area <- (area-a-sumar ? ?area)
  ?suma <- (suma ?total)
  =>
  (retract ?suma ?nueva-area)
  (assert (suma (+ ?total ?area))))

(defrule fin
  (not (area-a-sumar ? ?))
  (suma ?total)
  =>
  (printout t "La suma es " ?total crlf))
```

Control de la ejecución

- Ejemplo de fases de un problema:
  - Lectura de datos.
  - Detección de problemas.
- Técnicas de control:
  - Control empotrado en las reglas.
  - Control usando prioridades.
  - Control usando reglas.
  - Control usando módulos.

Control empotrado en las reglas

- Distinguir las fases del problema usando hechos de control:
  - `(fase lectura-de-datos)`
  - `(fase deteccion-de-problemas)`
- Todas las reglas de una misma fase tienen entre sus condiciones el hecho de control correspondiente
- Las reglas que finalizan una fase tienen que eliminar el hecho de control de dicha fase y añadir el de la siguiente
- Inconvenientes del control empotrado en las reglas:
  - Identificación de los hechos de control.
  - Dificultad para precisar la conclusión de cada fase.

Control empotrado en las reglas

```
(deffacts inicio
  (fase lectura-de-datos)
  (dispositivos C1 C2 C3))

(defrule lectura-de-datos
  (fase lectura-de-datos)
  ?h <- (dispositivos ?id $?res)
  (not (dato ?id ?val))
  =>
  (retract ?h)
  (assert (dispositivos $?res)
          (dato ?id (valor ?id))))

(defrule final-de-lectura-de-datos
  ?h1 <- (fase lectura-de-datos)
  ?h2 <- (dispositivos)
  =>
  (retract ?h1 ?h2)
  (assert (dispositivos C1 C2 C3)
          (fase deteccion-de-problemas)))
```

Control empotrado en las reglas

```
(defrule deteccion-de-problemas-1
  (fase deteccion-de-problemas)
  ?h <- (dato ?id ?val&:(evenp ?val))
  =>
  (retract ?h)
  (printout t "Problemas en el dispositivo " ?id t))

(defrule deteccion-de-problemas-2
  (fase deteccion-de-problemas)
  ?h <- (dato ?id ?val&:(oddp ?val))
  =>
  (retract ?h)
  (printout t "Sin problemas en el dispositivo " ?id t))

(defrule final-deteccion-de-problemas
  ?h <- (fase deteccion-de-problemas)
  (not (dato ? ?))
  =>
  (retract ?h)
  (assert (fase lectura-de-datos)))
```

Prioridades

- Sintaxis:

```
(defrule <nombre>
  (declare (salience <numero>))
  <condicion>*
  =>
  <accion>*)
```

- Valores:
  - Mínimo: -10000
  - Máximo: 10000
  - Defecto: 0
- Ventajas:
  - No es necesario precisar la conclusión de cada fase
- Inconvenientes:
  - Tendencia a abusar de las prioridades.
  - Contradicción con el objetivo de los sistemas basados en reglas.
  - Las fases se pueden mezclar.
  - Dificultad para comenzar otra vez el ciclo.

Control usando prioridades

```
(deffacts inicio
  (dispositivos C1 C2 C3))

(defrule lectura-de-datos
  (declare (salience 30))
  ?h <- (dispositivos $? ?id $?)
  (not (dato ?id ?val))
  =>
  (assert (dato ?id (random 1 100))))

(defrule deteccion-de-problemas-1
  (declare (salience 20))
  ?h <- (dato ?id ?val&:(evenp ?val))
  =>
  (retract ?h)
  (printout t "Problemas en el dispositivo " ?id t))

(defrule deteccion-de-problemas-2
  (declare (salience 20))
  ?h <- (dato ?id ?val&:(oddp ?val))
  =>
  (retract ?h)
  (printout t "Sin problemas en el dispositivo " ?id t))
```

Reglas de control

- Reglas específicas para controlar las fases de desarrollo
  - Se utilizan hechos de control para distinguir las fases
  - Se definen reglas para cambiar las fases con baja prioridad
- Ventajas
  - Separación entre reglas de control y reglas de proceso
  - Acorde con la arquitectura de referencia CommonKADS
  - No es necesario precisar la conclusión de cada fase

Control usando reglas

- Una regla para cada cambio de fase

```
(deffacts inicio
  (fase lectura-de-datos)
  (dispositivos C1 C2 C3))

(defrule lectura-a-deteccion
  (declare (salience -10))
  ?fase <- (fase lectura-de-datos)
  =>
  (retract ?fase)
  (assert (fase deteccion-de-problemas)))

(defrule deteccion-a-lectura
  (declare (salience -10))
  ?fase <- (fase deteccion-de-problemas)
  =>
  (retract ?fase)
  (assert (fase lectura-de-datos)))
```

Control usando reglas

- Una regla para cada cambio de fase

```
(defrule lectura-de-datos
  (fase lectura-de-datos)
  ?h <- (dispositivos $? ?id $?)
  (not (dato ?id ?val))
  =>
  (assert (dato ?id (random 1 100))))

(defrule deteccion-de-problemas-1
  (fase deteccion-de-problemas)
  ?h <- (dato ?id ?val&:(evenp ?val))
  =>
  (retract ?h)
  (printout t "Problemas en el dispositivo " ?id t))

(defrule deteccion-de-problemas-2
  (fase deteccion-de-problemas)
  ?h <- (dato ?id ?val&:(oddp ?val))
  =>
  (retract ?h)
  (printout t "Sin problemas en el dispositivo " ?id t))
```

Control usando reglas

- Una regla para todos los cambios de fase

```
(deffacts control
  (fase lectura-de-datos)
  (siguiente-fase lectura-de-datos deteccion-de-problemas)
  (siguiente-fase deteccion-de-problemas lectura-de-datos))

(defrule cambio-de-fase
  (declare (salience -10))
  ?fase <- (fase ?actual)
  (siguiente-fase ?actual ?siguiente)
  =>
  (retract ?fase)
  (assert (fase ?siguiente)))
```

Control usando reglas

- Cambio de fase en secuencia

```
(deffacts control
  (fase lectura-de-datos)
  (sucesion-de-fases lectura-de-datos
                     deteccion-de-problemas))

(defrule cambio-de-fase
  (declare (salience -10))
  ?fase <- (fase ?actual)
  (sucesion-de-fases ?siguiente $?resto)
  =>
  (retract ?fase)
  (assert (fase ?siguiente))
  (assert (sucesion-de-fases $?resto ?siguiente)))
```

Módulos

- Grupos de constructores (reglas, hechos, funciones, ...) con agenda propia
- Sólo un módulo está “activo” en cada momento, los restantes están “latentes”
- El módulo por defecto es MAIN y es el primero en construirse
- Sintaxis:

```
(defmodule <nombre>
  (export <elemento>)*
  (import <modulo> <elemento>)*)

<elemento> :== ?ALL |
               ?NONE |
               <constructor> ?ALL |
               <constructor> ?NONE |
               <constructor> <nombre>+

<constructor> :== deftemplate | defclass |
                  defglobal | deffunction | defgeneric
```

Comunicación entre módulos

- Lectura secuencial de ficheros:
  - Sólo se puede exportar a módulos sucesivos
  - Sólo se puede importar de módulos anteriores
  - El módulo MAIN puede exportar a cualquier otro pero no puede importar de ninguno
- Solución más general: exportarlo todo e importarlo todo de los módulos anteriores

```
(defmodule DETECCION
  (export ?ALL))

(defmodule AISLAMIENTO
  (export ?ALL))

(defmodule RECUPERACION
  (import DETECCION ?ALL)
  (import AISLAMIENTO ?ALL))
```

Definición de constructores en módulos

- Indicar delante del nombre del constructor el nombre del módulo
- Los constructores pueden hacer uso de cualquier elemento definido en el propio módulo o importado de otro módulo
- Módulo `DETECCION`

```
(deftemplate DETECCION::fallo
  (slot componente))

(deffacts DETECCION::inicio
  (fallo (componente A)))

(defrule DETECCION::deteccion
  (fallo (componente A | C))
  =>)
```

Definición de constructores en módulos

- Módulo **AISLAMIENTO**

```
(deftemplate AISLAMIENTO::posible-fallo
  (slot componente))

(deffacts AISLAMIENTO::inicio
  (posible-fallo (componente B)))

(defrule AISLAMIENTO::aislamiento
  (posible-fallo (componente B | D))
  =>)
```

- Módulo **RECUPERACION**

```
(deffacts RECUPERACION::inicio
  (fallo (componente C))
  (posible-fallo (componente D)))

(defrule RECUPERACION::recuperacion
  (fallo (componente A | C))
  (posible-fallo (componente B | D))
  =>)
```

Memorias de trabajo

- Visualizar la memoria de trabajo de un módulo:

```
CLIPS> (facts DETECCION)
f-1     (fallo (componente A))
f-3     (fallo (componente C))
For a total of 2 facts.
CLIPS> (facts AISLAMIENTO)
f-2     (posible-fallo (componente B))
f-4     (posible-fallo (componente D))
For a total of 2 facts.
CLIPS> (facts RECUPERACION)
f-1     (fallo (componente A))
f-2     (posible-fallo (componente B))
f-3     (fallo (componente C))
f-4     (posible-fallo (componente D))
For a total of 4 facts.
```

Memorias de trabajo

- Visualizar la memoria de trabajo de todos los módulos:

```
CLIPS> (facts *)
f-0     (initial-fact)
f-1     (fallo (componente A))
f-2     (posible-fallo (componente B))
f-3     (fallo (componente C))
f-4     (posible-fallo (componente D))
```

Agendas

- Visualizar la agenda de un módulo:

```
CLIPS> (agenda DETECCION)
0      deteccion: f-3
0      deteccion: f-1
For a total of 2 activations.
CLIPS> (agenda AISLAMIENTO)
0      aislamiento: f-4
0      aislamiento: f-2
For a total of 2 activations.
CLIPS> (agenda RECUPERACION)
0      recuperacion: f-1,f-4
0      recuperacion: f-1,f-2
0      recuperacion: f-3,f-4
0      recuperacion: f-3,f-2
For a total of 4 activations.
```

Agendas

- Visualizar la agenda de todos los módulos:

```
CLIPS> (agenda *)
MAIN:
DETECCION:
   0      deteccion: f-3
   0      deteccion: f-1
AISLAMIENTO:
   0      aislamiento: f-4
   0      aislamiento: f-2
RECUPERACION:
   0      recuperacion: f-1,f-4
   0      recuperacion: f-3,f-4
   0      recuperacion: f-3,f-2
   0      recuperacion: f-1,f-2
For a total of 8 activations.
```

Pila de módulos

- Estructura que establece la secuencia de módulos a ejecutar
  - Al iniciar el sistema de producción el módulo MAIN se coloca como primer módulo
  - Se pueden colocar otros módulos con el comando **`focus`**

```
CLIPS> (reset)
CLIPS> (watch rules)
CLIPS> (run)
CLIPS> (focus DETECCION)
TRUE
CLIPS> (run)
FIRE    1 deteccion: f-3
FIRE    2 deteccion: f-1
```

Pila de módulos

- El comando `(list-focus-stack)` permite ver el estado de la pila de módulos

```
CLIPS> (reset)
CLIPS> (focus RECUPERACION)
TRUE
CLIPS> (focus AISLAMIENTO)
TRUE
CLIPS> (list-focus-stack)
AISLAMIENTO
RECUPERACION
MAIN
CLIPS> (run)
FIRE    1 aislamiento: f-4
FIRE    2 aislamiento: f-2
FIRE    3 recuperacion: f-1,f-4
FIRE    4 recuperacion: f-3,f-4
FIRE    5 recuperacion: f-3,f-2
FIRE    6 recuperacion: f-1,f-2
```

Pila de módulos

- Se pueden apilar varios módulos en una misma instrucción `focus`

```
CLIPS> (reset)
CLIPS> (focus AISLAMIENTO RECUPERACION)
TRUE
CLIPS> (list-focus-stack)
AISLAMIENTO
RECUPERACION
MAIN
CLIPS> (run)
FIRE    1 aislamiento: f-4
FIRE    2 aislamiento: f-2
FIRE    3 recuperacion: f-1,f-4
FIRE    4 recuperacion: f-3,f-4
FIRE    5 recuperacion: f-3,f-2
FIRE    6 recuperacion: f-1,f-2
```

Pila de módulos

- Se puede apilar varias veces un mismo módulo, aunque no de forma consecutiva

```
CLIPS> (reset)
CLIPS> (focus DETECCION DETECCION AISLAMIENTO DETECCION)
TRUE
CLIPS> (list-focus-stack)
DETECCION
AISLAMIENTO
DETECCION
MAIN
CLIPS> (run)
FIRE    1 deteccion: f-3
FIRE    2 deteccion: f-1
FIRE    3 aislamiento: f-4
FIRE    4 aislamiento: f-2
```

Pila de módulos

- El comando `pop-focus` elimina el primer módulo de la pila
  - Este comando se puede usar en una regla como acción

```
CLIPS> (reset)
CLIPS> (focus DETECCION AISLAMIENTO RECUPERACION)
TRUE
CLIPS> (list-focus-stack)
DETECCION
AISLAMIENTO
RECUPERACION
MAIN
CLIPS> (run 2)
FIRE    1 deteccion: f-3
FIRE    2 deteccion: f-1
CLIPS> (pop-focus)
AISLAMIENTO
CLIPS> (list-focus-stack)
RECUPERACION
MAIN
CLIPS> (run)
FIRE    1 recuperacion: f-1,f-4
FIRE    2 recuperacion: f-3,f-4
FIRE    3 recuperacion: f-3,f-2
FIRE    4 recuperacion: f-1,f-2
```

Pila de módulos

- El comando `(watch focus)` permite visualizar como cambian los elementos en la pila de módulos

```
CLIPS> (reset)
CLIPS> (watch focus)
CLIPS> (focus DETECCION AISLAMIENTO RECUPERACION)
==> Focus RECUPERACION from MAIN
==> Focus AISLAMIENTO from RECUPERACION
==> Focus DETECCION from AISLAMIENTO
TRUE
CLIPS> (run)
FIRE    1 deteccion: f-3
FIRE    2 deteccion: f-1
<== Focus DETECCION to AISLAMIENTO
FIRE    3 aislamiento: f-4
FIRE    4 aislamiento: f-2
<== Focus AISLAMIENTO to RECUPERACION
FIRE    5 recuperacion: f-1,f-4
FIRE    6 recuperacion: f-3,f-4
FIRE    7 recuperacion: f-3,f-2
FIRE    8 recuperacion: f-1,f-2
<== Focus RECUPERACION to MAIN
<== Focus MAIN
```

Acciones sobre la pila de módulos

- El comando `focus` se puede usar como acción en una regla

```
(defmodule MAIN
  (export ?ALL))

(defrule MAIN::inicio
  =>
  (focus DETECCION AISLAMIENTO RECUPERACION))
```

Acciones sobre la pila de módulos

```
CLIPS> (reset)
CLIPS> (watch focus)
CLIPS> (run)
FIRE    1 inicio: f-0
==> Focus RECUPERACION from MAIN
==> Focus AISLAMIENTO from RECUPERACION
==> Focus DETECCION from AISLAMIENTO
FIRE    2 deteccion: f-3
FIRE    3 deteccion: f-1
<== Focus DETECCION to AISLAMIENTO
FIRE    4 aislamiento: f-4
FIRE    5 aislamiento: f-2
<== Focus AISLAMIENTO to RECUPERACION
FIRE    6 recuperacion: f-1,f-4
FIRE    7 recuperacion: f-3,f-4
FIRE    8 recuperacion: f-3,f-2
FIRE    9 recuperacion: f-1,f-2
<== Focus RECUPERACION to MAIN
<== Focus MAIN
```

Acciones sobre la pila de módulos

- El comando `pop-focus` se puede usar como acción en una regla

```
(defrule DETECCION::deteccion
  (fallo (componente A | C))
  =>
  (pop-focus))

(defrule AISLAMIENTO::aislamiento
  (posible-fallo (componente B | D))
  =>
  (pop-focus))

(defrule RECUPERACION::recuperacion
  (fallo (componente A | C))
  (posible-fallo (componente B | D))
  =>
  (pop-focus))
```

Acciones sobre la pila de módulos

```
CLIPS> (reset)
CLIPS> (watch focus)
CLIPS> (run)
FIRE    1 inicio: f-0
==> Focus RECUPERACION from MAIN
==> Focus AISLAMIENTO from RECUPERACION
==> Focus DETECCION from AISLAMIENTO
FIRE    2 deteccion: f-3
<== Focus DETECCION to AISLAMIENTO
FIRE    3 aislamiento: f-4
<== Focus AISLAMIENTO to RECUPERACION
FIRE    4 recuperacion: f-1,f-4
<== Focus RECUPERACION to MAIN
<== Focus MAIN
```

Acciones sobre la pila de módulos

- La declaración `(declare (auto-focus TRUE))` en una regla coloca el módulo al que pertenece en la pila de módulos al activarse dicha regla

```
(defmodule RECUPERACION
  (import DETECCION ?ALL)
  (import AISLAMIENTO ?ALL)
  (export ?ALL))

(defrule RECUPERACION::recuperacion
  (fallo (componente A | C))
  (posible-fallo (componente B | D))
  =>
  (assert (alarma)))

(defmodule ALARMA
  (import RECUPERACION ?ALL))

(defrule ALARMA::alarma
  (declare (auto-focus TRUE))
  (alarma)
  =>)
```

Acciones sobre la pila de módulos

```
CLIPS> (reset)
CLIPS> (watch focus)
CLIPS> (run)
FIRE    1 inicio: f-0
==> Focus RECUPERACION from MAIN
==> Focus AISLAMIENTO from RECUPERACION
==> Focus DETECCION from AISLAMIENTO
FIRE    2 deteccion: f-3
FIRE    3 deteccion: f-1
<== Focus DETECCION to AISLAMIENTO
FIRE    4 aislamiento: f-4
FIRE    5 aislamiento: f-2
<== Focus AISLAMIENTO to RECUPERACION
FIRE    6 recuperacion: f-1,f-4
==> Focus ALARMA from RECUPERACION
FIRE    7 alarma: f-5
<== Focus ALARMA to RECUPERACION
FIRE    8 recuperacion: f-3,f-4
FIRE    9 recuperacion: f-3,f-2
FIRE   10 recuperacion: f-1,f-2
<== Focus RECUPERACION to MAIN
<== Focus MAIN
```

Control usando módulos

```
(defmodule MAIN
  (export ?ALL))

(deffacts MAIN::inicio
  (ciclo)
  (dispositivos C1 C2 C3))

(defrule MAIN::control
  ?h <- (ciclo)
  =>
  (retract ?h)
  (assert (ciclo))
  (focus LECTURA DETECCION))
```

Control usando módulos

```
(defmodule LECTURA
  (import MAIN ?ALL)
  (export ?ALL))

(defrule LECTURA::lectura-de-datos
  ?h <- (dispositivos $? ?id $?)
  (not (dato ?id ?val))
  =>
  (assert (dato ?id (random 1 100))))
```

Control usando módulos

```
(defmodule DETECCION
  (import LECTURA ?ALL))

(defrule DETECCION::deteccion-de-problemas-1
  ?h <- (dato ?id ?val&:(evenp ?val))
  =>
  (retract ?h)
  (printout t "Problemas en el dispositivo " ?id t))

(defrule DETECCION::deteccion-de-problemas-2
  ?h <- (dato ?id ?val&:(oddp ?val))
  =>
  (retract ?h)
  (printout t "Sin problemas en el dispositivo " ?id t))
```

Bibliografía

- Giarratano, J.C. y Riley, G. “Expert Systems Principles and Programming (2nd ed.)” (PWS Pub. Co., 1994)
  - Cap. 7 - 12
- Giarratano, J.C. “CLIPS 6.0 User’s Guide”