11 APRENDIZAJE INTENCIONAL

La librería `JILDT` extiende las capacidades de los agentes a través de acciones internas y planes, así como una arquitectura de agente.

11.1 ACCIONES INTERNAS Y FUNCIONES MATEMÁTICAS

Como es normal, las acciones internas se implementan en Java y son herederas de la clase `DefaultInternalAction`, que a su vez implementa la interfaz `InternalAction`. Estas acciones se organizan en los paquetes `jildt`, `jildt.tilde` y `jildt.tilde.math`. Por ello, a cada una de las acciones se le debe anteponer el prefijo correspondiente.

11.1.1 Paquete `jildt`

Las acciones internas que conforman este paquete permiten ejecutar funciones generales relacionadas con el aprendizaje, como obtener información necesaria para el ejecutar el algoritmo de aprendizaje; o incorporar lo aprendido en el programa del agente aprendiz. La Figura 11.1 presenta el diagrama de clases[1] de las acciones internas implementadas en `jildt`. A saber:

- `jildt.displayTree(Tree,Mode)`. Determina el modo de visualización de un árbol lógico que unifica con la variable `Tree`. Si `Mode` unifica con `console` el árbol se despliega en la consola de Jason; si unifica con `gui`, se despliega en una ventana gráfica; `both` despliega el árbol de las dos formas. La Figura **??** ilustra el resultado de esta configuración.
- `jildt.getContext(PlanLabel,Ctxt)`. Unifica la variable `Ctxt` con el contexto del plan etiquetado como `PlanLabel`. El contexto se obtiene como una instancia de la clase `LogicalFormula`. Cabe recordar que los planes con contexto vacío, en realidad tienen contexto `true`.
- `jildt.setContext(PlanLabel,Ctxt)`. Esta acción interna substituye el contexto del plan cuya etiqueta es `PlanLabel`, por un nuevo contexto `Ctxt`.
- `jildt.getCurrentBels(Bs)`. Unifica la variable `Bs` con la lista de las creencias actuales del agente. Se excluyen las creencias que tienen que

1 En adelante, se emplea una notación UML estándar, donde los atributos y métodos públicos se denotan con el símbolo + , los privados se denotan por - y los protegidos con #. Los métodos subrayados refieren métodos estáticos. Las relaciones de herencia de clase e implementación de interfaz se denotan con una flecha con línea continua y línea punteada respectivamente, dirigidas hacia la clase de la cual se hereda o la interfaz que se implementará. Las clases sombreadas representan aquellas que están incluidas dentro de la librería JILDT.

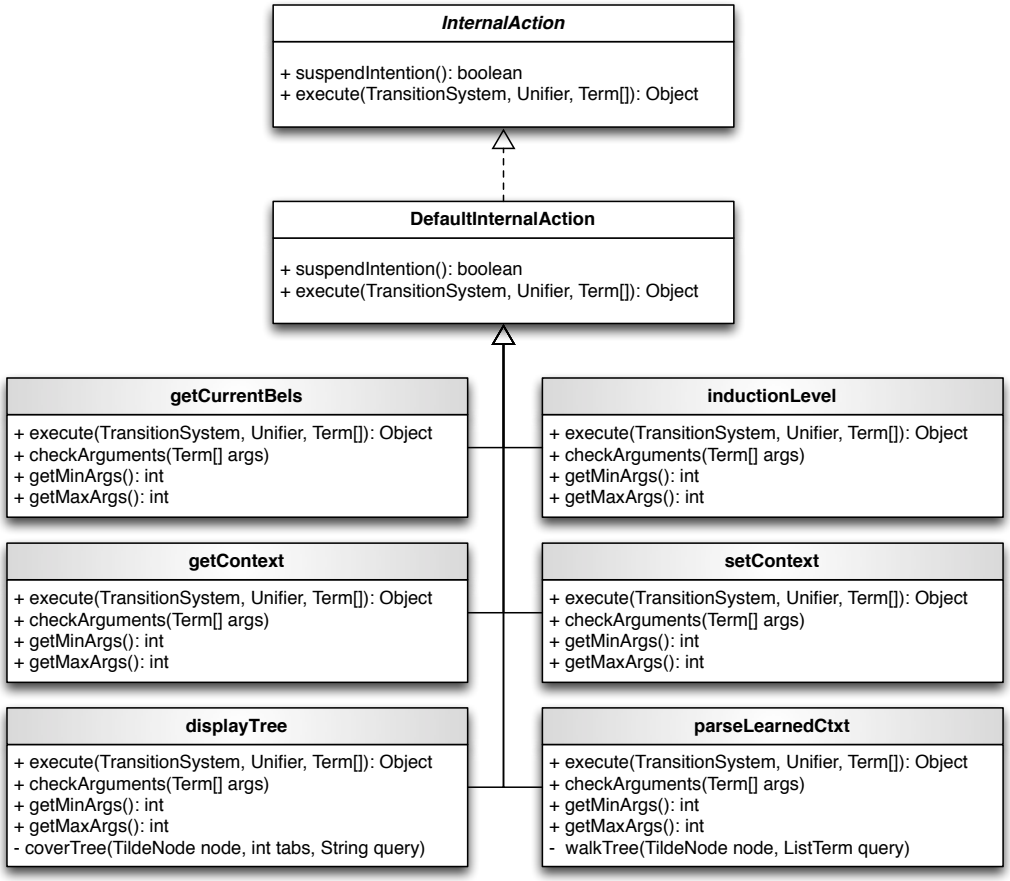


Figura 11.1: Diagrama de clase de las acciones internas en el paquete `jildt`.

ver con el proceso de aprendizaje (aquellas que inician con el prefijo `jildt`). El usuario puede excluir también las creencia que considere irrelevantes para el aprendizaje, mediante la creencia `jildt_settings(excludeBels,[`b_0`,...,`b donde las b_i son los símbolos de las creencias que se desea excluir. Las reglas también son excluidas, pues éstas constituyen, para el caso del aprendizaje, lo que se conoce como el conocimiento previo de del agente.

- `jildt.inductionLevel(Lvl)`. Unifica la variable `Lvl` con el nivel al que se está ejecutando la inducción. A nivel `java`, el proceso de aprendizaje se ejecuta a través de una acción interna; a nivel `agentSpeak` el proceso de aprendizaje se ejecuta a través de un conjunto de planes y diversas acciones internas definidas en el paquete `jildt.tilde`. La configuración se lleva a cabo con la creencia `jildt_settings(inductionLevel,Lvl)`.

- `jildt.parseLearnedCtxt(Tree,LC)`. Unifica la variable `LC` con una representación basada en listas del contexto de un plan construído a partir del árbol lógico que unifica con `Tree`. El resultado se expresa como una lista de listas de literales. Cada lista es una conjunción de literales, mientras que la lista de listas representa una disyunción.

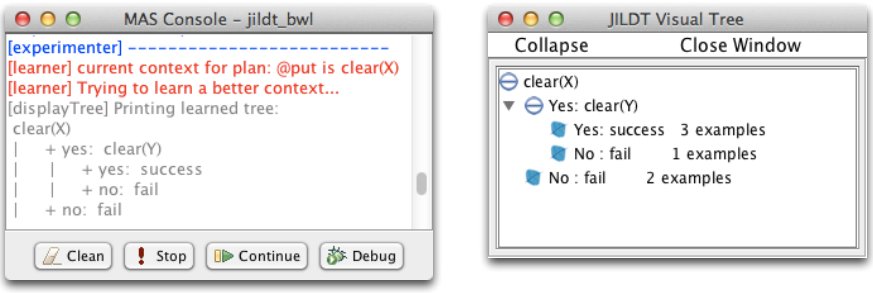


Figura 11.2: Visualización de un árbol lógico de decisión en consola (izquierda) y en una ventana gráfica (derecha).

11.1.2 Paquete `jildt.tilde`

Las acciones internas que conforman este paquete permiten ejecutar funciones directamente relacionadas con la construcción de árboles lógicos de decisión. La Figura 11.3 presenta el diagrama de clases de las acciones internas implementadas en el paquete `jildt.tilde`, a saber:

- `execTilde(PlanLabel,Tree)`. Ejecuta el algoritmo TILDE para construir un árbol lógico de decisión. La variable `PlanLabel` unifica con la etiqueta del plan para el cual se ejecutará el algoritmo de aprendizaje. La variable `Tree` unifica con el árbol aprendido.

- `findExamples(PlanLabel,Exs)`. Unifica `Exs` con una lista de indices de los ejemplos de entrenamiento asociados al plan cuya etiqueta es `PlanLabel`. Los ejemplos son almacenados en una base de creencias personalizada que indiza las literales contenidas en ésta. Se utilizan los índices para facilitar el manejo de los ejemplos de entrenamiento entre los planes que hacen uso de estos, ya que enviar listas de literales `jildt_example/3` puede llegar a ser muy computacionalmente pesado, debido a la longitud de este tipo de literales. Sin embargo, como trabajo futuro se pretende dejar la opción abierta a utilizar la representación que mejor se adecue al problema (índices o literales).

- `initialQuery(PlanLabel,Qi)`. Construye la consulta inicial Q_i para construir un árbol lógico de decisión. La consulta inicial es una lista cuyo único miembro es una literal *jildt_intend*/1, cuyo argumento es el evento disparador del plan etiquetado como `PlanLabel`. La consulta inicial fue diseñada para vincular lógicamente las variables en el evento disparador del plan con las del árbol lógico por construir, mediante el sesgo del lenguaje. Se considera en versiones posteriores tomar en cuenta las variables que aparecen en la cabeza del plan y no solo en su evento disparador.

- `languageBias(Exs,Lb)`. Permite computar el sesgo del lenguaje a partir la lista de índices de ejemplos de entrenamiento que unifica con la variable `Exs`. Hay dos opciones para definir el sesgo del lenguaje, que es posible configurar vía la creencia `jildt_settings(biasMode,Mode)`. Si la variable `Mode` unifica con `automatic`, se computan todas las posibles combinaciones de la directiva `jildt_rm\1` que se pueden formar

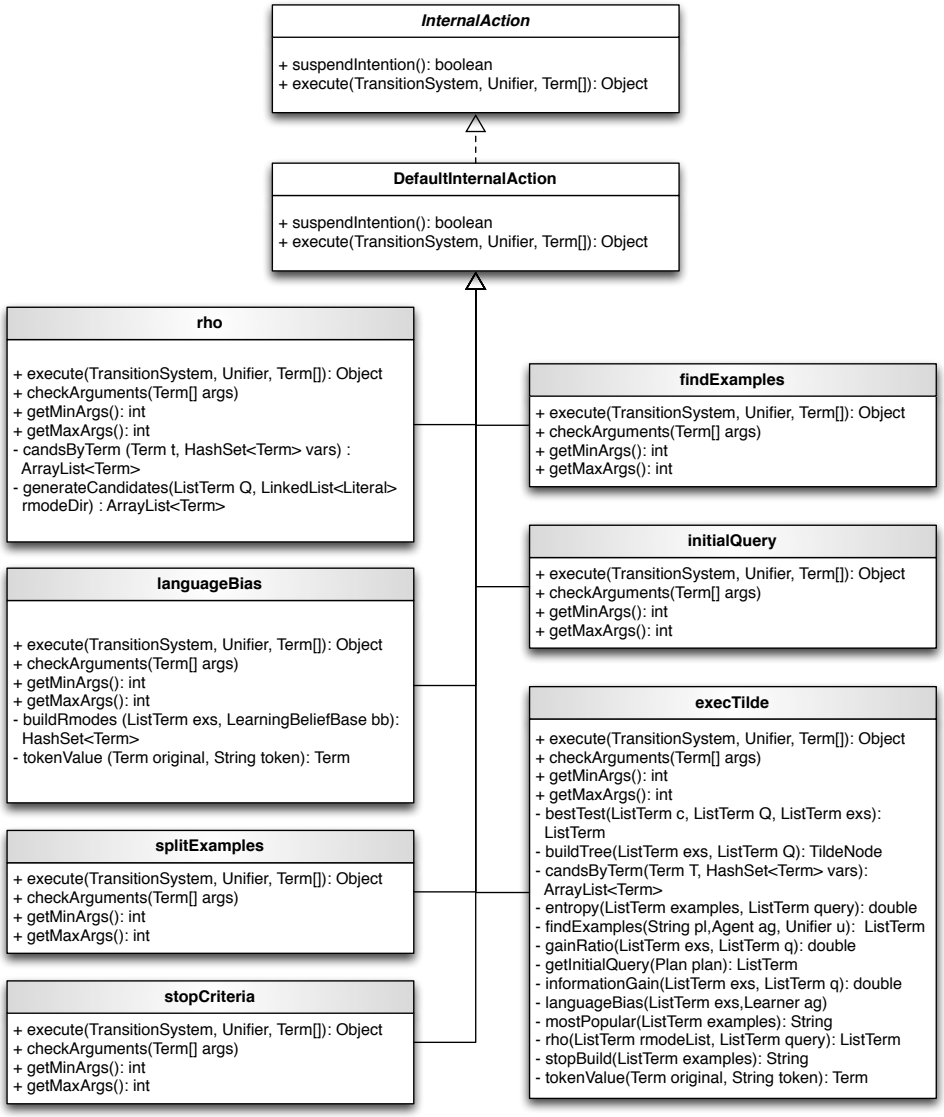


Figura 11.3: Diagrama de clase de las acciones internas en `jildt.tilde`

con las literales usadas en los ejemplos y su aridad. Si la variable `Mode` unifica con `manual` las directivas `jildt_rm\1` no son computadas automáticamente. En este caso el programador las define en el programa del agente. La variable `Lb` unifica con la lista de directivas `jildt_rm\1`, independientemente de como estas fueron computadas. Pero, observen que mientras que en la opción manual, los *rmodes* del sesgo del lenguaje ya están incluidos en la base de creencias del agente; en el caso automático esto no sucede. Los *rmodes* en la lista `Lb` deberán ser agregados por el programador a la base de creencias del agente.

- `rho(Q,Cns)`. La variable `Cns` unifica con una lista de candidatos a formar parte del árbol lógico, a partir del query `Q` y los *rmodes* generados con la acción interna anterior o definidos manualmente por el usuario.

- `splitExamples(Exs,Qb,Sp)`. La variable `Sp` unifica con una lista de la lista de los índices de los ejemplos de entrenamiento en `Exs` que satisfa-

cen la consulta `Qb` y otra lista de los que no la satisfacen. Un uso alternativo de esta acción interna es `jildt.tilde.splitExamples(Exs,Qb,[S,F])`, donde la variable `S` unifica con los ejemplos que satisfacen la consulta y `F` con los que no.

- `stopCriteria(Exs,percentage(P),Class)`. Esta acción define un criterio de paro para el algoritmo TILDE. La variable `Class` unifica con la clase mayoritaría de los ejemplos de entrenamiento cuyos índices están en `Exs`, si al menos `P` porciento de los ejemplos pertenece a una misma clase.

11.1.3 Paquete `jildt.tilde.math`

JILDT también implementa las funciones matemáticas asociadas con el algoritmo de aprendizaje TILDE. Estas funciones se encuentran implementadas en el paquete `jildt.tilde.math` y son herederas de la clase `DefaultArithFunction`, la cual a su vez implementa la interfaz `ArithFunction`. Las funciones matemáticas sobrescriben el método `evaluate`, el cual es llamado por el intérprete del agente para ejecutar la acción interna.

A diferencia de las acciones internas, las funciones matemáticas regresan valores reales `Double`. Esto último presenta una ventaja ya que permite ejecutar una función matemática desde el código en Java de otra función. Por ejemplo, la función de ganancia de información requiere ejecutar mas de una vez la función de entropía. Para aprovechar aún más esta ventaja, las funciones matemáticas implementadas en la librería JILDT implementan un método estático llamado `calculate`, el cual ejecuta el método `evaluate` de la función, y facilita su ejecución desde cualquier otra clase implementada en Java.En Jason, el resultado de una función matemática debe asignarse a una variable, por ejemplo, `GR = jildt.math.gainRatio(Exs,Qb)`.

Finalmente, para poder utilizar las funciones matemáticas, éstas deben definirse al principio del código del agente con el enunciado `register_function("package.fun")`, sin embargo, un agente aprendiz implementa un método que se encarga de esto. La Figura 11.4 muestra el diagrama de clases de las funciones matemáticas implementadas.

11.2 EXTENSIÓN DE PLANES

Los planes originales de los agentes pueden ser extendidos de modo que puedan ejecutar acciones relacionadas con el proceso de aprendizaje. La extensión de planes se lleva a cabo desde una directiva de pre-procesamiento llamada `LearnablePlans`, que se encuentra en del paquete `jildt` e implementa la interfaz `Directive` (Ver figura 11.5).
Las directivas de pre-procesamiento se utilizan para pasar algunas instrucciones al intérprete que no están relacionados con la semántica del lenguaje, sino que son meramente sintácticas. La directiva `LearnablePlans` es usada para modificar los planes que deseamos sean sujetos del aprendiza-

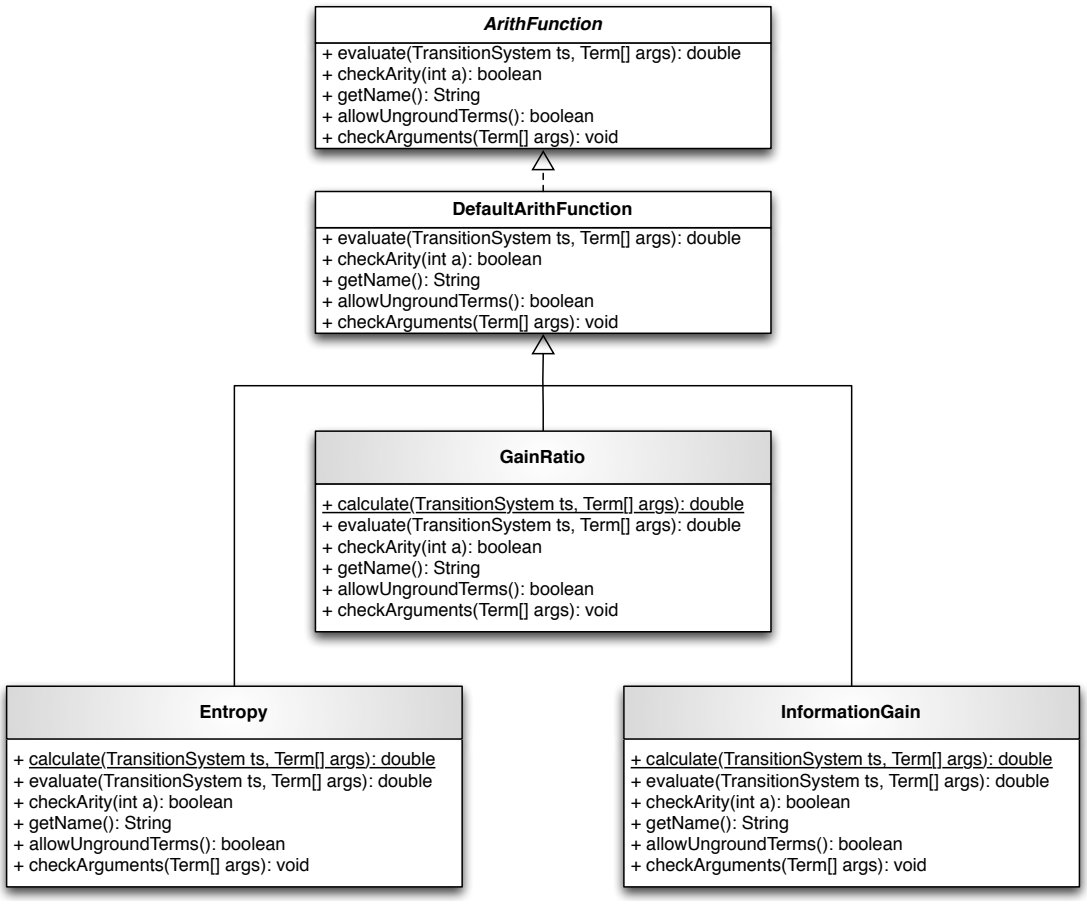


Figura 11.4: Diagrama de clase de las funciones matemáticas en `jildt.tilde.math`.

je intencional. El cuadro 11.1 muestra la sintáxis de esta directiva de pre-procesamiento[2].
Para ejemplificar en qué consiste la extensión de planes, se introduce un programa de agente para el mundo de los bloques, el cual se muestra en el cuadro 11.2. En este ejemplo, solo los planes etiquetados como `@put_succCase` y `@put_failCase` serán extendidos, ya que se encuentran dentro del alcan-

2 El nombre de la clase es *LearnablePlans*, sin embargo, para utilizar esta directiva dentro del código *AgentSpeak(L)* es necesario definirla en el archivo de configuración del SMA *(*.mas2j)* con la sentencia **directives: learnablePlans = jildt.LearnablePlans**, por ello, en adelante se refiere al uso de esta directiva como *learnablePlans*.

```
1 {begin learnablePlans}
2    +!p0;
3    -!p1;
4    ...
5    +!pn;
6 {end}
```

Cuadro 11.1: Sintaxis del uso de la directiva de pre-procesamiento `jildt.LearnablePlans`

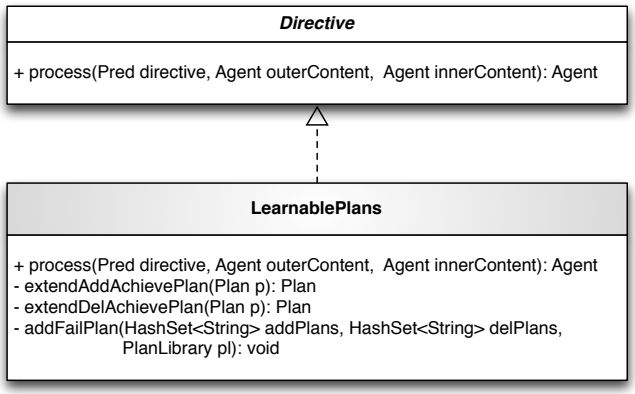


Figura 11.5: Diagrama de clase de la directiva de pre-procesamiento *jildt.LearnablePlans*.

ce de la directiva `learnablePlans`; mientras que el plan etiquetado como `@put_failCaseNonApplicable` no se extiende.

Dos tipos de extensiones han sido implementadas, una para los planes que se ejecutan al agregar una meta de logro `(+!)` y otra para los planes que se ejecutan cuando fallan estos planes `(-!)`. En el primer caso, el plan original es extendido (ver el cuadro 11.3) en tres partes: la primera parte, o preprocesamiento (líneas 3-6), consiste en obtener información relevante sobre el mundo actual del agente, antes de ejecutar el plan original. Primero se obtiene el deseo actual del agente, usando la acción interna `.desire(Des)`, después se obtienen las creencias del agente al momento de iniciar la ejecución del plan, usando la acción interna `jildt.getCurrentBeliefs(Bs)`. El deseo y las creencias actuales del agente formarán parte del ejemplo de entrenamiento y son concatenadas en una lista. Una vez que se obtiene el estado BDI del agente se agrega la creencia `jildt_tagPlan(put_succCase)` para indicar cual es el plan que se está intentando ejecutar. La segunda parte introduce el cuerpo original del plan (líneas 7-9). Por último, la tercera parte (líneas 10-11), que se ejecuta únicamente si el plan original concluye con éxito, se elimina la creencia `jildt_tagPlan(put_succCase)`, y se agrega un ejemplo de entrenamiento etiquetado como `success`, indicando que una vez que se intentó ejecutar el plan `put_succCase`, teniendo el deseo `Des` y conociendo las creencias `Bs`, la ejecución del plan fue satisfactoria (`success`). En el caso de los planes que se ejecutan al fallar un plan de logro `(-!)`, la extensión se realiza de la siguiente manera (ver el cuadro 11.4): al contexto del plan original se añade una consulta para ejecutar este plan únicamente cuando el error se produjo por un fallo en una acción interna `(ia_failed)`, en una acción del plan `(action_failed)`, en una consulta `(ask_failed)` o en una restricción `(constraint_failed)`. En el cuerpo del plan, se obtienen el deseo actual del agente y las creencias del agente al momento de iniciar la ejecución del plan, para formar la lista de creencias que formarán parte del ejemplo de entrenamiento (líneas 3-5). Después de esto, se consulta qué plan estaba siendo ejecutado cuando se produjo el error(línea 7) y se ejecuta un plan de aprendizaje (línea 8), del cual aprenderá si es posible, nuevas razones para adoptar el plan etiquetado como `Tag`. Por último, al finalizar

```
/* Initial beliefs and rules */
clear(X) :- not(on(_,X)).
clear(table).

/* Plans */
{begin learnablePlans}
@put_succCase
+!put(X,Y) : true <-
  move(X,Y);
  .print("Yeah, I did the task successfully");
  .send(experimenter,tell,experiment(done)).

@put_failCase
-!put(X,Y) : true <-
  .send(experimenter,tell,experiment(done)).
{end}

@put_failCaseNonApplicable
-!put(X,Y)[error(Error)] : .member(Error,[
    ↪ no_applicable,no_relevant,no_option,wrong_arguments,unknown]) <-
  .print("Plan +!put produced an irrelevant failure.");
  .send(experimenter,tell,[non_applicable(put),experiment(done)]).
```

Cuadro 11.2: Programa de agente Jason para el mundo de los bloques.

```
@put_succCase
+!put(X,Y) : true <-
  .desire(Des);
  jildt.getCurrentBels(Bs);
  .concat([intend(Des)], Bs, Train);
  +jildt_tagPlan(put_succCase);
  move(X,Y);
  .print("Yeah, I did the task succesfully");
  .send(experimenter,tell,experiment(done)).
  -jildt_tagPlan(put_succCase);
  +jildt_example(put_succCase, Train, success).
```

Cuadro 11.3: Extensión del plan *put_succCase* del cuadro 11.2.

```
1  @put_failCase
2  -!put(X,Y)[error(Error)] : .member(Error,[ia_failed, action_failed, ask_failed,
       ↪ constraint_failed]) <-
3    .desire(Des);
4    jildt.getCurrentBels(Bs);
5    .concat([intend(Des)], Bs, Train);
6    ?jildt_tagPlan(Tag);
7    +jildt_example(Tag, Train, fail);
8    !learning(Tag, Tree);
9    -jildt_tagPlan(Tag);
10   .send(experimenter,tell,experiment(done)).
```

Cuadro 11.4: Extensión del plan *put_failCase* del cuadro 11.2.

```
1  @learning
2  +!learning(P,Tree): true <-
3    .print("Trying to learn a better context...");
4    !executeTilde(P,Tree);
5    jildt.displayTree(Tree, both);
6    jildt.parseLearnedCtxt(Tree, LC);
7    .print("Learned context for ",P," is ", LC);
8    jildt.setContext(P, LC).
```

Cuadro 11.5: Plan de aprendizaje.

el proceso de aprendizaje, elimina la creencia `jildt_tagPlan(Tag)`(línea 9) y ejecuta las acciones que conformaban el plan de fallo original (línea 10).

11.2.1 Planes de aprendizaje

Como se muestra en el cuadro 11.4, un plan que responde a un evento de fallo (`-!`) intenta ejecutar el plan de aprendizaje `!learning(Tag, Tree)` para aprender un nuevo contexto para el plan que estaba siendo ejecutado al momento de producirse el fallo, definido en la variable `Tag`. A continuación se explica el funcionamiento de los planes implementados para ejecutar el proceso de aprendizaje. Los planes de aprendizaje se encuentran definidos en el archivo de agente `learningPlans.asl`, en el paquete `jildt`. Sin embargo, puede definirse el origen de estos planes usando la literal `jildt_settings(learningPlansSrc, <PATH>)` en el programa de agente.

El Cuadro 11.5 muestra el plan de aprendizaje ejecutado desde un plan de fallo extendido. Primero se ejecuta el plan `executeTilde(P, Tree)`(línea 4) para construir el árbol lógico de decisión relacionado con el fallo producido por el plan etiquetado como `P`. El árbol computado es unificado con la variable `Tree`. Una vez computado el árbol lógico de decisión, se visualiza usando la acción interna `jildt.displayTree(Tree, both)`. Por último se obtiene un nuevo contexto a partir de las ramas del árbol computado que llevan a un nodo hoja etiquetado como `success`, y se sustituye el contexto del plan que provocó el fallo con el contexto aprendido.

La construcción de un árbol lógico de decisión puede ejecutarse en dos niveles distintos de programación: Como una acción interna programada en Java y como un conjunto de planes programados en Jason. Para configurar el nivel de inducción, se declara una literal `jildt_settings(inductionLevel, Lvl)`, donde `Lvl` unifica con `java` o `agentSpeak`, según sea el caso. Por defec-

```
1  @execTilde_agSpeak
2  +!executeTilde(P, Tree) : jildt.inductionLevel(agentSpeak) <-
3    jildt.tilde.findExamples(P, Exs);
4    jildt.tilde.languageBias(Exs, Lb);
5    for (.member(Rm,Lb)){+Rm;};
6    jildt.tilde.initialQuery(P, Qi);
7    !buildTree(Exs, Qi, Tree).
8
9  @execTilde_java
10 +!executeTilde(P, Tree) : jildt.inductionLevel(java) <-
11   jildt.tilde.execTilde(P,Tree).
```

Cuadro 11.6: Planes de inducción de TILDE

to, y por motivos de eficiencia, un agente aprendiz ejecuta el algoritmo de aprendizaje en un nivel Java.

El Cuadro 11.6 muestra los planes definidos para la construcción de un árbol lógico de decisión. El plan etiquetado como `@execTilde_agSpeak`(líneas 1-7) ejecuta un plan para inducir la construcción del árbol si el si el nivel de inducción es `agentSpeak`, en caso contrario, se ejecuta el plan etiquetado como `@execTilde_java`(líneas 9-11), el cual construye el árbol desde la acción interna `jildt.tilde.execTilde`.

11.2.2 Construcción de Árboles Lógicos de Decisión en *AgentSpeak(L)*

El plan etiquetado como `@execTilde_agSpeak` en el cuadro 11.6 construye un árbol lógico como se describe a continuación. Primero, se buscan los ejemplos de entrenamiento relacionados con el plan `P`(línea 3). Una vez que se obtienen los ejemplos de entrenamiento, se genera y agrega el sesgo del lenguaje (líneas 4-5). El siguiente paso es formar la consulta inicial `Qi` (línea 6), para empezar a construir el árbol lógico de decisión con las entradas generadas anteriormente.

La construcción del árbol se realiza recursivamente usando los planes mostrados en el cuadro 11.7. El plan `@buildTree_stop` detiene la construcción del árbol cuando un criterio de paro se cumple. En este caso, la variable `Tree` unifica con una lista que representa un nodo hoja, con su respectiva etiqueta de clase y el total de ejemplos que pertenecen a ésta (línea 4). En caso de no cumplirse el criterio de paro, se ejecuta el plan `@buildTree_recursive`. Primero genera los candidatos a formar parte del árbol (línea 8) y selecciona el mejor de ellos a través del plan `bestTest(Exs, Q, Cns, Bcn)`(Cuadro 11.8). Una vez que se obtiene el mejor candidato, se construye una nueva consulta `Qb` agregando el mejor candidato a la consulta inicial (línea 10) y se dividen los ejemplos de entrenamiento entre aquellos que satisfacen `Qb` y aquellos que no (línea 11). Dos árboles se construyen y formaran los nodos izquierdo y derecho del árbol (líneas 12-13). El nodo `izquierdo` se forma a partir de los ejemplos que satisfacen la consulta `Qb` y se envía como consulta inicial a `Qb`; el nodo `derecho` se forma a partir de los ejemplos que no satisfacen `Qb` y recibe como consulta inicial a `Q`. La variable `Tree` unifica con una lista formada por la literal del nodo y sus nodos izquierdo y derecho.
El plan `bestTest` computa el coeficiente `Gain Ratio` para todos los candidatos recibidos en la variable `Cns`. La literal `jildt_bestTest` permite almacenar

```
@buildTree_stop
+!buildTree(Exs,_,Tree): jildt.tilde.stopCriteria(Exs,percentage(100),Class) <-
  .length(Exs, L);
  Tree = [Class, L].

@buildTree_recursive
+!buildTree(Exs, Q,Tree): true <-
  jildt.tilde.rho(Q, Cns);
  !bestTest(Exs, Q, Cns, Bcn);
  .concat(Q, [Bcn], Qb);
  jildt.tilde.splitExamples(Exs, Qb, [ExsLeft, ExsRight]);
  !buildTree(ExsLeft, Qb, Left);
  !buildTree(ExsRight, Q, Right);
  Tree = [Bcn, Left, Right].
```

Cuadro 11.7: Planes para construir un árbol lógico de decisión.

```
@bestTest
+!bestTest(Exs, Q, Cns, Bcn): true <-
  +jildt_bestTest(true, 0);
  for (.member(Cn,Cns)){
    .concat(Q, [Cn], Qb);
    GR = jildt.math.gainRatio(Exs,Qb);
    ?jildt_bestTest(Temp, Max);
    if (GR > Max) {
      -jildt_bestTest(Temp, Max);
      +jildt_bestTest(Cn, GR);
    };
  };
  ?jildt_bestTest(Bcn, Max);
  -jildt_bestTest(Bcn, Max).
```

Cuadro 11.8: Plan de selección del mejor candidato.

temporalmente la información sobre el mejor candidato durante la búsqueda. La variable `Bcn` unifica con el candidato que maximice su valor de `Gain Ratio`.

11.3 CLASE DE AGENTE `learner`

Bordini, Hübner y Wooldridge [16] describen a un agente, desde el punto de vista de un intérprete *AgentSpeak(L)* extendido, como un conjunto de creencias (`Bs`), un conjunto de planes (`Ps`), algunas funciones de selección definidas por el usuario y la función de confianza (una relación "socialmente aceptable" para los mensajes recibidos), funciones usadas en el ciclo de razonamiento (por ejemplo, la función de actualización de creencias (`BUF`) y la función de revisión de creencias (`BRF`)), y una instancia de la clase `Circumstance`, la cual incluye eventos pendientes, intenciones y otras estructuras necesarias durante la interpretación del agente. La implementación por defecto de estas funciones está codificada en una clase llamada `Agent`, la cual ha sido personalizada extendiendo sus funciones básicas para aprender nuevas razones para adoptar planes toda vez que haya fallado en la ejecución de estos y se encuentra en el paquete `jildt.agent`. La figura 11.6 muestra el diagrama de clases de un agente aprendiz `Learner`. Por razones de presentación, solo se

muestran los atributos que son propios de la clase de agente `Learner` y los métodos que fueron sobrescritos, sobrecargados o que son propios de esta clase de agente (a excepción de los `get` y `set`).

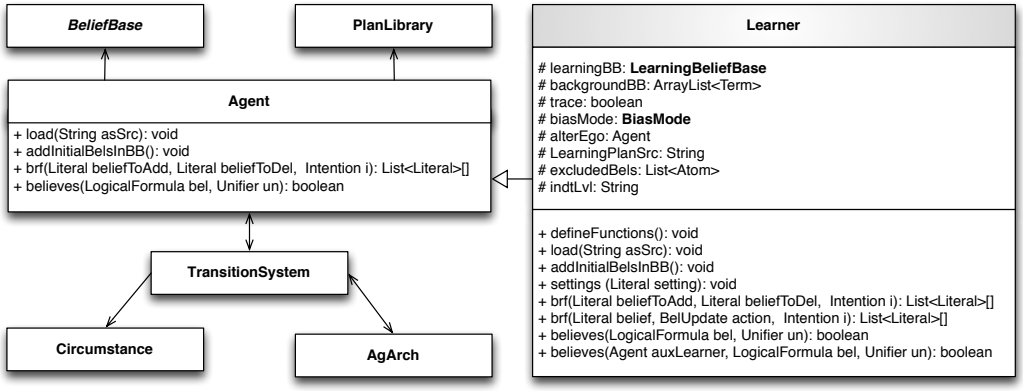


Figura 11.6: Diagrama de clase de un agente aprendiz `Learner`, Adaptado de Bordini, Hübner y Wooldridge [16]

La clase de agente `Learner` especifica tres tipos de datos enumerados: `BelUpdate`, `BiasMode` y `SettingsAtt`, los cuales se explicarán a lo largo de esta sección. A diferencia de un agente convencional, un agente aprendiz está compuesto por atributos que son usados en el proceso de aprendizaje y se enlistan a continuación:

- `trace`. Variable `booleana` que indica si la construcción del árbol es trazada en consola o no. Se configura con la creencia `jildt_settings(trace,Val)`, donde `Val` unifica con `true` o `false` (valor por default).
- `excludedBels`. Es una lista de átomos $(List < Atom >)$ que almacena los símbolos de las creencias que no formaran parte de los ejemplos de entrenamiento. Se configura desde el programa de agente agregando la creencia `jildt_settings(excludeBels,Bels)`, donde `Bels` es la lista de símbolos, por default vacía.
- `indtLvl`. Una variable de tipo cadena que almacena el nivel de programación en el que se ejecutará la inducción de árboles lógicos de decisión. Se configura agregando la creencia `jildt_settings(inductionLevel, Lvl)`, donde `Lvl` unifica con `agentSpeak` o `java` (valor por default).
- `LearningPlanSrc`. Este atributo define la ruta del archivo origen donde se definen los planes de aprendizaje. Se puede configurar agregando la creencia `jildt_settings(learningPlansSrc, Src)`, donde `Src` es la ruta del archivo. La ruta por defecto es `jildt/learningPlans.asl`.
- `biasMode`. Variable del tipo `Learner.BiasMode` que indica el modo en que el agente formará y utilizará el sesgo del lenguaje. Se configura con la creencia `jildt_settings(biasMode, BM)`, donde `BM` unifica con `manual` o `automatic` (valor por default).
- `learningBB`. Este atributo es una instancia de una Base de Creencias personalizada en la libreria JILDT, permite separar las creencias del

agente relacionadas con el proceso de aprendizaje, de las creencias que el agente tiene de su entorno y problemática para la que fue implementado.

- `backgroundBB`. Un lista dinámica de términos ($ArrayList < Term >$) que apunta a las reglas del agente. Representan el conocimiento general del agente, por lo que tener una lista que apunte directamente a este conocimiento mejora la eficiencia del agente, evitando una búsqueda exhaustiva de las reglas en la base de creencias del agente.

- `alterEgo`. Una instancia de la clase `Agent` que apunta a la base de creencias de aprendizaje `learningBB`. El objetivo de este atributo es auxiliar en la búsqueda de una literal de aprendizaje, ya que por defecto, los métodos en un agente común de Jason buscan las creencias únicamente en la base de creencias implementada por defecto (Ver Figura 11.7).

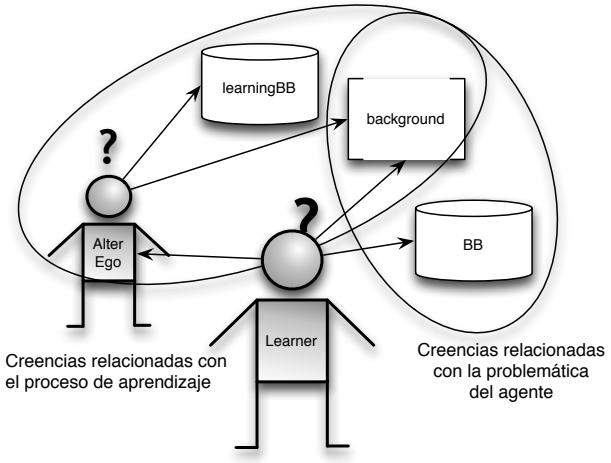


Figura 11.7: Modo de operación de un agente aprendiz con respecto a la consulta de sus creencias.

La clase de agente `Learner` hereda la mayoría de los métodos definidos en la clase `Agent`, pero también sobre escribe algunos otros. El método `load` es sobrescrito para cargar los planes de aprendizaje, una vez se haya ejecutado el método `load(String asSrc)` de la clase `Agent`; el método `addInitialBelsInBB()` es sobrescrito para agregar las reglas que forman el conocimiento general del agente a la lista `backgroundBB`. Esta asociación se hace únicamente al agregar las creencias iniciales, ya que los agentes Jason no agregan reglas dinámicamente.

Dos métodos fueron sobrescritos y sobrecargados para manejar las creencias relacionadas con el aprendizaje. El método de revisión de creencias `brf` se sobrescribe para preguntar si la creencia que se agregará o eliminará es una creencia relacionada con el aprendizaje. Si es así, ejecuta el método sobrecargado `brf(Literal belief, BelUpdate action, Intention i)`, pasando como argumento una variable del tipo enumerado `Learner.BelUpdate`, que indica si la variable se agregará o eliminará de la base de creencias de

aprendizaje; de no tratarse de una creencia de aprendizaje, se ejecuta el método `brf(Literal beliefToAdd, Literal beliefToDel, Intention i)` definido en la clase `Agent`. El otro método sobrescrito y sobrecargado es el método `believes`, el cual pregunta si una creencia es consecuencia lógica de las creencias del agente. Si se pregunta por una creencia de aprendizaje, se ejecuta el método sobrecargado `believes(Agent auxLearner, LogicalFormula bel, Unifier un)`, el cual recibe como argumento la variable `AlterEgo`, que apunta a la base de creencias de aprendizaje, que es donde busca la creencia consultada; en caso de no tratarse de una creencia de aprendizaje, se ejecuta el método `believes(LogicalFormula bel, Unifier un)` implementado en la clase `Agent`.

El método `defineFunctions()` define las funciones matemáticas implementadas en `jildt.tilde.math` y es ejecutado al inicio del método `load`. El método `settings (Literal setting)` guarda la configuracion recibida en su argumento. Se ejecuta desde el método `brf` cada vez que una literal del tipo `jildt_settings/2` es añadida. El tipo de dato enumerado `Learner.SettingsAtt` es usado en este método para asociar el tipo de configuración en su argumento con el atributo correspondiente en la clase de agente `Learner`. Por último, se implementan los métodos `setters` y `getters` correspondientes a cada atributo.

11.3.1 LearningBeliefBase

Como se mencionó anteriormente, las creencias del agente relacionadas con el aprendizaje son separadas de las creencias que el agente tiene sobre la problemática para la cual fue diseñado, para ello se extiende el estado mental de un agente `Learner` para incluir una base de creencias de aprendizaje. Esta base de creencias incluye principalmente ejemplos de entrenamiento `(jildt_example/3)`, directivas que forman el sesgo del lenguaje `(jildt_rm/1)` y configuraciones `(jildt_setings/2)`. Para el caso de los ejemplos de entrenamiento, el uso de las literales puede llegar a ser complicado, ya que el tamaño de estas puede ser bastante amplio. Para ello se ha implementado la clase `LearningBeliefBase`, la cual define una base de creencias personalizada, que hereda sus métodos de la clase `DefaultBeliefBase` que a su vez implementa la interfaz `BeliefBase`. La Figura 11.8 muestra el diagrama de la clase `LearningBeliefBase` y sus clases auxiliares `BelieveEntry` y `LiteralWrapper`, mismas que se encuentran en el paquete `jildt.bb`.
Una base de creencias contiene una instancia de la clase `Map` que mapea un predicado[3] con una instancia de la clase `BelieveEntry` (ver Figura 11.9). Esta clase define dos tipos de mapeos, uno entre la `Literal` y su índice, y otro entre el índice y la `Literal`. Los índices son relativos, a cada predicado en la base de creencias corresponde un mapeo de índices diferente. Además, se agrega una anotación que indica cuantas veces se ha agregado la misma creencia. Esta última característica nos abre la puerta criterios probabilísticos de desempate al obtener el mejor candidato en la construcción de un árbol lógico de decisión, y forma parte de los trabajos a futuro de este proyecto.

3 Un `predicado` es una estructura formada por el funtor y la aridad de una literal, por ejemplo, el predicado de `on(X,Y)` es `on/2`.

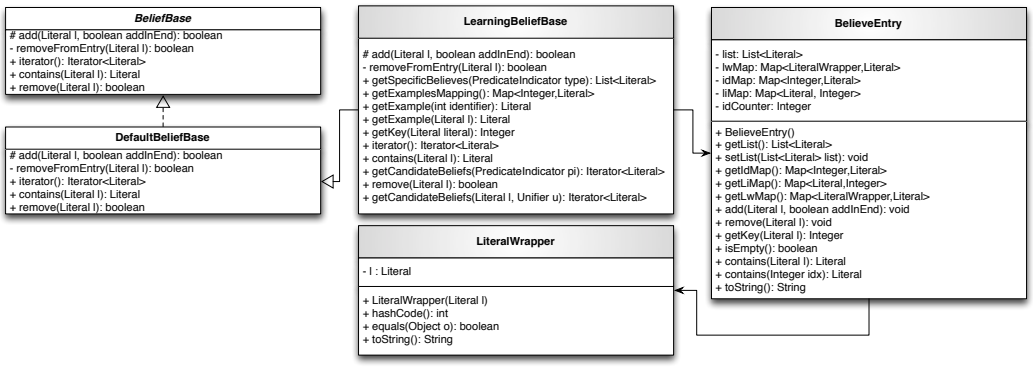


Figura 11.8: Diagrama de clases de `LearningBeliefBase` y sus clases auxiliares.

La clase `LiteralWrapper` únicamente sirve como auxiliar para buscar una `Literal` a partir de otra.

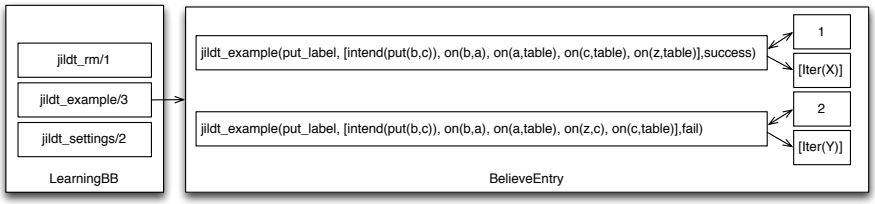


Figura 11.9: Composición de una entrada dentro de una base de creencias de aprendizaje.

11.3.2 Clase de agente `SingleMindedLearner`

Se ha implementado la clase de agente `SingleMindedLearner`, la cual extiende de la funcionalidad de un agente `Learner` para adoptar una estrategia de compromiso racional (`SingleMinded commitment`) a partir del aprendizaje obtenido a través de la inducción de árboles lógicos de decisión. Esta implementación forma el caso de estudio de este proyecto[4].

La clase de agentes `Learner` permite definir agentes capaces de redefinir el contexto de sus planes toda vez que haya ocurrido un fallo en la ejecución de un plan. De este modo, este tipo de agentes aprende a reconsiderar sus acciones antes de adoptar una intención futura, cada vez que haya fallado en la ejecución de un plan. De modo muy similar, los agentes definidos como instancias de la clase `SingleMindedLearner` son capaces, además de redefinir el contexto de sus planes, de adquirir conocimiento general que expresa cuándo es racional abandonar una intención una vez que percibe que no podrá finalizarla con éxito. Esta clase de agente implementa una estrategia de compromiso racional, mediante reglas de abandono, donde el cuerpo de estas reglas se obtiene de las ramas del árbol lógico de decisión que llevan a un nodo hoja etiquetado como `fail`.

4 Al no ser parte del núcleo de la librería JILDT, la clase de agente y las funcionalidades implementadas se encuentran definidas en un paquete llamado `sml`, incluido en la distribución de JILDT.

```
1 drop(put(A,B)) :- .intend(put(X,Y)) & clear(A) & not(clear(B)).
2 drop(put(A,B)) :- .intend(put(X,Y)) & not (clear(A)).
```

Cuadro 11.9: Reglas de abandono formadas a partir del árbol mostrado en la figura **??**.

```
1 @dropPlan
2 +dropIntention(I) : true <-
3   .print("Wow!! I'm sorry, I have to abandon my intention");
4   .drop_intention(I).
```

Cuadro 11.10: Plan de abandono de intenciones en el agente `singleMindedLearner`.

El comportamiento de un agente tipo `singleMindedLearner` se rige por las siguientes reglas:

1. Si existe un plan aplicable, y el contexto del plan es consecuencia lógica del conjunto de creencias del agente, entonces ejecuta el plan seleccionado, y si la ejecución es satisfactoria, agrega un ejemplo de entrenamiento etiquetado como `success`.

2. Si al momento de ejecutar el plan seleccionado ocurre un fallo, entonces se ejecuta un plan de aprendizaje, y en caso de haber aprendido un contexto diferente, lo redefine y agrega al menos una regla de abandono que indica cuándo es racional abandonar una intención. Una regla de abandono esta formada de la siguiente manera: La cabeza es una literal `drop(I)`, donde `I` unifica con la intención actual del agente; el cuerpo está formado por la consulta `.intend(I)` y la conjunción de literales que forman el camino recorrido desde el nodo raíz de un árbol hasta un nodo hoja etiquetado como `fail`. El Cuadro 11.9 muestra las reglas de abandono obtenidas a partir del árbol lógico de decisión mostrado en la figura **??**.

3. Si posteriormente, al momento de tratar de ejecutar una intención I, el agente percibe que la literal `drop(I)` es consecuencia lógica de su conjunto de creencias, entonces dispara el evento `+dropIntention(I)`, que ejecuta el plan etiquetado como `@dropPlan` (Ver Cuadro 11.10). Este plan fuerza al agente a abandonar la intención `I`, usando la acción interna `.drop_intention(I)` (línea 4) que es parte de las acciones internas incluidas en la distribución de `Jason`.

El mecanismo anterior se ejemplifica en la Figura 11.10. En este caso, el ciclo de razonamiento es modificado para percibir cuando es racional abandonar una intención ($SelInt_3$).

$$(\textbf{SelInt}_3) \quad \frac{Ag_{bs} \models drop(I)}{\langle ag, C, M, T, SelInt\rangle \rightarrow \langle ag, C', M, T, ClrInt\rangle}$$

donde: $\mathcal{C}'_{\mathcal{I}} = \mathcal{C}'_{\mathcal{I}} / \{\mathcal{I}\}$

Los planes de aprendizaje de un agente definido como instancia de la clase `SingleMindedLearner` se encuentran en el archivo `smLearnPlans.asl` en el paquete `sml`. Esta configuración es posible definiendo la literal `jildt_settings(learningPlansSrc,`

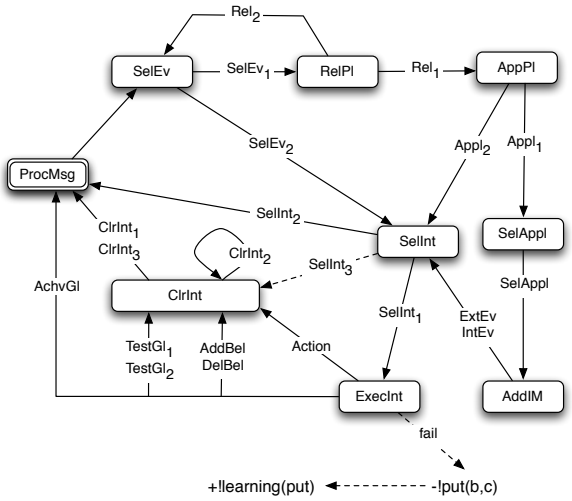


Figura 11.10: Modo operacional de un agente tipo `singleMindedLearner`.

```
1 @learning
2 +!learning(P,Tree): true <-
3   .print("Trying to learn a better context...");
4   !executeTilde(P,Tree);
5   jildt.parseLearnedCtxt(Tree,LC);
6   jildt.displayTree(Tree, gui);
7   .print("Learned context for ",P," is ", LC);
8   jildt.setContext(P, LC);
9   sml.addDropRule(P,Tree).
```

Cuadro 11.11: Plan de aprendizaje para un agente `SingleMindedLearner`.

"`../sml/smLearnPlans.asl`") en el programa de agente. El Cuadro 11.11 muestra el plan de aprendizaje modificado para un agente `SingleMindedLearner`, el cual agrega la acción interna `sml.addDropRule(P,Tree)`(línea 9), implementada para agregar las reglas de abandono. La variable `P` indica el plan para el que se ejecuto el proceso de aprendizaje, y la variable `Tree` es el árbol computado. El resto de los métodos descritos en la sección 11.2 se mantienen iguales.

11.4 OTRAS CLASES Y FUNCIONES

Además de las clases que se han mencionado a lo largo de este capítulo, la librería JILDT cuenta con otras clases. El paquete `jildt`, aparte de las acciones internas y el programa de agente que define los planes aprendizaje, implementa dos clases: `Functions` y `TildeNode`(Ver Figura 11.11).
La clase **Functions** contiene funciones estáticas de uso general, las cuales son utilizadas en más de una acción interna, de ahı' su implementación como métodos estáticos. La clase **TildeNode** implementa un `nodo` de árbol. Los métodos definidos en esta clase permiten consultar si un nodo es un nodo interno `(inner node)` o un nodo hoja `(leaf node)`. Dos métodos estáticos permiten convertir una instancia de la clase `TildeNode` en una instancia de

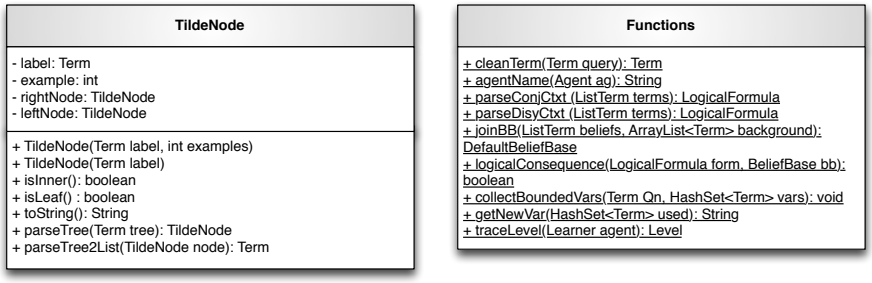


Figura 11.11: Diagrama de clases de `TildeNode` y `Functions`.

la clase `Term` y viceversa, para garantizar una buena integración entre la representación de un árbol en Java y su representación en `AgentSpeak(L)`.

Por último, el paquete `jildt.gui` implementa una clase de interfaz gráfica de usuario, que funge como herramienta de visualización y despliega una ventana que muestra el árbol computado como el presentado en la Figura 11.2 (derecha).

11.5 APRENDIENDO A PINTAR

Consideremos un agente `pintor` que vive en un ambiente rejilla, donde su tarea consiste en pintar el piso de su ambiente sin pararse sobre el césped que está en el borde, o los mosaicos que ya ha pintado de rojo (Ver Figura 11.12).

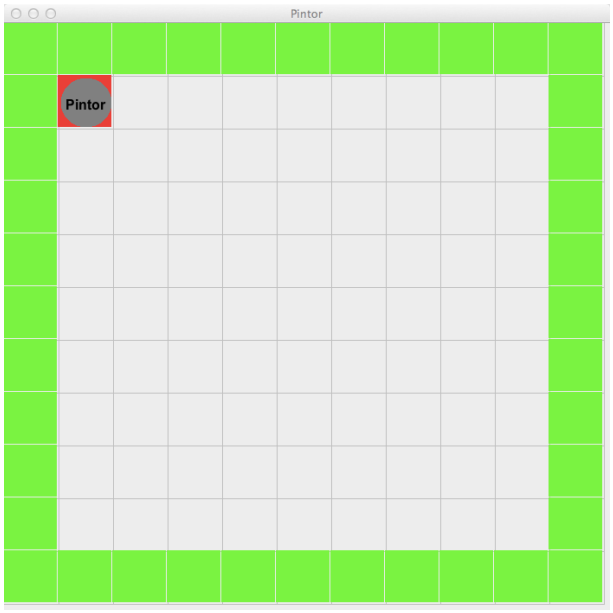


Figura 11.12: El agente `pintor` en su ambiente.

El SMA que usaremos define cuatro agentes, aunque de momento los agentes que aprenden están comentados:

```
1 /*
2     Jason Project
3
4     -- created on septiembre 25, 2012
```

```
*/

MAS painters {
  infrastructure: Centralised
  environment: Environment.PainterEnv(10,10, 1, 1)
  agents:
    // Uncomment the agent that will execute the experiment:
    // Comment the others.
    master;
    //optimal agentClass jildt.agent.Learner;
    //random agentClass painter.PainterLearner;

    classpath: "../../../lib/jildt.jar";
  aslSourcePath:"src/asl";
}
```

La directiva de compilación `learnablePlans` permite especificar cuales son los planes que se pueden revisar vía el aprendizaje. El `classpath` apunta a la implementación de `jildt`. Finalmente, `aslSourcePath` permite editar el SMA en el ambiente de desarrollo eclipse. El ambiente donde estará situado el agente `pintor` es una rejilla de 10×10 mosaicos. La posición inicial del agente es la esquina superior izquierda $(1,1)$, tal y como lo especifica `environment` (La definición del ambiente `pintorEnv` se revisa más adelante en la sección 11.5.3):

11.5.1 El Agente Optimo

El agente `master` es un chico listo que sabe los contextos de sus planes, es decir, es un programa de agente correcto para su ambiente. De hecho establece el desempeño óptimo para los agentes en este SMA: con este programa es posible pintar todos los mosaicos del ambiente. Su definición es como sigue:

```
/* Initial goals */
!go(_).

/* Plans */
@go
+!go(Dir): floor(Dir) & unpainted(Dir) <-
  paint;
  //.print("Moving to ", Dir);
  move(Dir);
  //.print("Success");
  !!go(_).

@go_failCase
-!go(_): true <-
  !!go(_).
```

Primero, observen que la representación de este agente es **proposicional**. El agente puede ejecutar cuatro acciones: `movDer`, `movIzq`, `movAba`, y `movArr`, que mueven al agente una casilla a la derecha, a la izquierda, abajo ó arriba, respectivamente. Con estas acciones se pueden definir cuatro planes para explorar el ambiente: `@derecha`, `@abajo`, `@izquierda` y `@arriba`. La condición de estos planes es que si el agente quiere moverse en cierta dirección, debe percibir que hay piso en esa dirección (no hay pasto) y el piso en esa dirección no está pintado. Si el plan para `!mover` falla, simplemente se reporta el

fallo en el plan `@mover_failCase`. Si no hay planes aplicables para `!mover` no hay nada más que hacer, tal y como lo expresa el plan `@mover_final`.

Aunque no hay nada en este programa que diga que la meta del agente es pintar todos los cuadros del ambiente, este programa converge al óptimo. La cuestión ahora es si los contextos de estos planes se pueden aprender intencionalmente.

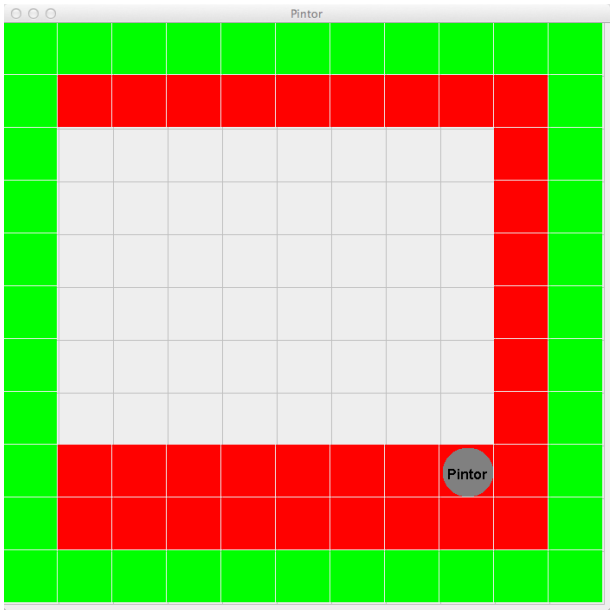
11.5.2 El Agente Aprendiz

La configuración de `jildt` nos indica que no se excluirá ninguna creencia en el proceso de aprendizaje; y se mostrará la traza de ejecución al correr el SMA. El agente tiene 4 planes para moverse y pinter, cuyos contextos no han sido definidos. Es de esperar que el agente aprenda dichos contextos a partir de su experiencia. Para ello, los planes que responden al evento `+!mover` han sido definidos dentro de la directiva `learnablePlans`. Todos estos planes hacen dos cosas, pintan el mosaico donde se encuentra el agente, mediante la acción externa `pintar`; se mueven en alguna dirección (derecha, izquierda, abajo y arriba, es el orden en que están definidos estos planes); y finalmente llaman a `!mover` recursivamente. El plan de fallo `-!mover`, simplemente despliega un mensaje de que la intención ha fallado.

La ejecución de `pintor` arroja un curioso resultado. El primer plan aplicable con respecto al evento `+!mover`, etiquetado como `@derecha`, tiene como contexto `true`, por lo que es seleccionado repetidamente, moviendo con éxito al agente hacía la derecha, hasta alcanzar la esquina superior derecha del ambiente. En ese momento la acción `movDer` del plan `@derecha` falla y el agente intenta aprender un nuevo contexto con tres ejemplos (recuerden que a diferencia de Prolog, los hechos repetidos de colapsan en una solo hecho al agregarlos a la base de creencias del agente). El nuevo contexto de este plan es `pisoderecha`, indicando que el plan será aplicable solo cuando hay piso accesible a la derecha del agente. El fallo es reportado por el plan de recuperación para `-!mover`.

En ese momento, el plan `@derecha` ya no es aplicable en la posición actual del agente, por lo que el plan `@abajo` es seleccionado como aplicable. El agente se moverá hasta la esquina inferior derecha donde fallará. En ese momento el agente aprenderá, como en el caso anterior, que `@abajo` tienen un nuevo contexto `pisoabajo`. A continuación el agente se moverá un lugar a la izquierda y a continuación intentará moverse a la derecha, porque el contexto del plan `@derecha` es verdadero. Solo que su intención fallará porque ese mosaico ya ha sido pintado. Nuevamente, un proceso de aprendizaje para el plan `@derecha` con cuatro ejemplos (los tres originales y el nuevo), es ejecutado. El nuevo contexto aprendido es `pisoabajo & pisoderecha`. Como no hay `pisoabajo` en la posición actual del agente, éste se moverá hacía la izquierda hasta llegar a la esquina inferior. Ahí se aprende que el contexto para el plan `@izquierda` es `pisoizquierda`. A continuación el agente se mueve hacía la derecha hasta que al intentar moverse al mosaico pintado cuando bajaba, el plan falla y se lanza un nuevo proceso de aprendizaje. El resultado es `sinpintarderecha & pisoderecha`. Al intentar moverse abajo el agente vuelve a fallar y aprende un nuevo contexto `pisoabajo & sinpintarderecha`.

Luego intentará moverse a la izquierda, fallando y aprendiendo del nuevo el contexto que ya sabía `pisoizquierda`. Este comportamiento se repite indefinidamente quedando el agente atascado en la situación siguiente:



Hay dos problemas asociados a este comportamiento:

- El orden de los planes en el programa del agente y los contextos aprendidos, determinan los ejemplos que serán colectados más adelante. Es decir, lo aprendido introduce un sesgo en la exploración del espacio de hipótesis posibles.
- Una vez en atascado, el agente es incapaz de aprender `sinpintarizquierda & pisoizquierda`.

11.5.3 El Ambiente de los Pintores

Este ambiente se define usando la estrategia modelo-vista-controlador. El controlador del ambiente es la clase `PintorEnv` implementada en Java, como sigue:

```
1  package Environment;
2
3
4  import jason.asSyntax.Atom;
5  import jason.asSyntax.Literal;
6  import jason.asSyntax.Structure;
7  import jason.environment.Environment;
8  import jason.environment.grid.Location;
9
10 import java.util.logging.Logger;
11
12 /** Simple Vacuum cleaning environment */
13 public class PainterEnv extends Environment {
14
15   private static final long serialVersionUID = 1L;
16   Logger logger = Logger.getLogger(PainterEnv.class.getName());
```

```
17
18  // Model reference
19  PainterModel model;
20  private int rows;
21  private int columns;
22
23  @Override
24  public void init(String[] args) {
25    if (args.length == 4 ){
26      int rows = Integer.parseInt(args[0]);
27      int columns = Integer.parseInt(args[1]);
28      int x = Integer.parseInt(args[2]);
29      int y = Integer.parseInt(args[3]);
30      this.rows = rows;
31      this.columns = columns;
32      model = new PainterModel(rows, columns, x, y);
33      PainterView view = new PainterView(model);
34      model.setView(view);
35      updatePercepts();
36    }else
37      System.out.println("Please indicate both the number of rows and columns,
            ↪ as the X,Y position of the agent");
38  }
39
40  private void updatePercepts(){
41    clearPercepts();
42    Location r1 = model.getAgPos(0);
43
44    // floor(Dir): Is added when a floor space is found in Dir. Grass is not
          ↪ floor.
45    // unpainted(Dir): Is added when an unpainted space in floor is found in Dir
          ↪ .
46
47    if (r1.x < this.columns-1)
48      addPercept(Literal.parseLiteral("floor(right)"));
49    if(r1.x >= this.columns-1  || !model.board[r1.y][r1.x+1])
50      addPercept(Literal.parseLiteral("unpainted(right)"));
51    if (r1.x >  1)
52      addPercept(Literal.parseLiteral("floor(left)"));
53    if(r1.x ==  0 || !model.board[r1.y][r1.x-1])
54      addPercept(Literal.parseLiteral("unpainted(left)"));
55    if (r1.y < this.rows-1)
56      addPercept(Literal.parseLiteral("floor(down)"));
57    if(r1.y >= this.rows-1 || !model.board[r1.y+1][r1.x])
58      addPercept(Literal.parseLiteral("unpainted(down)"));
59    if (r1.y >  1)
60      addPercept(Literal.parseLiteral("floor(up)"));
61    if(r1.y ==  0 || !model.board[r1.y-1][r1.x])
62      addPercept(Literal.parseLiteral("unpainted(up)"));
63  }
64
65  @Override
66  public synchronized boolean executeAction(String ag, Structure action) {
67    // For knowing if everything worked fine
68    boolean actResult = true;
69
70    if (action.getFunctor().equals("move")) {
71      PainterModel.Directions direction = PainterModel.Directions.valueOf(((Atom
            ↪ ) action.getTerm(0)).getFunctor());
72      switch (direction){
```

```
73        case right:
74          actResult = this.model.moveRight();
75          break;
76        case left:
77          actResult = this.model.moveLeft();
78          break;
79        case down:
80          actResult = this.model.moveDown();
81          break;
82        case up:
83          actResult = this.model.moveUp();
84          break;
85        }
86      }else if (action.getFunctor().equals("paint")) {
87        actResult = model.paint();
88      }else {
89        logger.info("Action "+action+" is not implemented!");
90        return false;
91      }
92
93      // Everything is fine, so percepts get updated
94      if(actResult)
95        updatePercepts();
96      try{
97        // Sleep the tread for doing slower the simulation.
98        Thread.sleep(200);
99      }catch(Exception e){}
100
101     return actResult;
102   }
103 }
```

El modelo del ambiente es definido como sigue:

```
1  package Environment;
2
3  import java.util.logging.Logger;
4
5  import jason.environment.grid.GridWorldModel;
6  import jason.environment.grid.Location;
7
8
9  public class PainterModel extends GridWorldModel{
10
11   private static final long serialVersionUID = 1L;
12   Logger logger = Logger.getLogger(PainterModel.class.getName());
13
14   // For knowing which boxes are painted
15   boolean[][] board;
16     private int rows;
17     private int columns;
18
19     enum Directions {right, left, down, up};
20
21     public PainterModel(int rows, int columns, int x, int y){
22       // Extend the board to add the grass around it
23       super(rows+1, columns+1, 1);
24         this.rows = rows;
25         this.columns = columns;
26         setAgPos(0, x, y);
27         board = new boolean[rows+1][columns+1];
```

```
28
29          //Grass is added
30          for(int i = 0; i <=rows; i++)
31            add(16, 0, i);
32          for(int i = 0; i <=rows; i++)
33              add(16, rows, i);
34          for(int i = 1; i <=columns; i++)
35              add(16, i, 0);
36          for(int i = 1; i <=columns; i++)
37              add(16, i, columns);
38      }
39
40      boolean moveRight(){
41        Location r1 = getAgPos(0);
42        // The action fails if there is no floor to the right, or the right box is
              ↪  painted.
43        if(r1.x+1 >= this.columns || board[r1.y][r1.x+1] || isOnGrass(r1.x, r1.y))
44            return false;
45        // Move and set in the new position
46        setAgPos(0, r1.x+1, r1.y);
47          return true;
48      }
49
50      boolean moveLeft(){
51          Location r1 = getAgPos(0);
52       // The action fails if there is no floor to the left, or the left box is
              ↪ painted.
53          if(r1.x-1 < 1 || board[r1.y][r1.x-1]|| isOnGrass(r1.x, r1.y))
54            return false;
55          // Move and set in the new position
56          setAgPos(0, r1.x-1, r1.y);
57          return true;
58      }
59
60      boolean moveDown(){
61        Location r1 = getAgPos(0);
62        // The action fails if there is no floor down, or the down box is painted.
63          if(r1.y +1 >= this.rows || board[r1.y+1][r1.x]|| isOnGrass(r1.x, r1.y))
64              return false;
65          // Move and set in the new position
66          setAgPos(0, r1.x, r1.y+1);
67          return true;
68      }
69
70      boolean moveUp(){
71        Location r1 = getAgPos(0);
72        // The action fails if there is no floor up, or the up box is painted.
73          if(r1.y-1 < 1 || board[r1.y-1][r1.x]|| isOnGrass(r1.x, r1.y))
74              return false;
75          // Move and set in the new position
76          setAgPos(0, r1.x, r1.y-1);
77          return true;
78      }
79
80      boolean paint(){
81        Location r1 = getAgPos(0);
82          board[r1.y][r1.x] = true;
83          add(32, r1.x, r1.y);
84          this.view.update();
85          // This action never fails
```

```
86          return true;
87      }
88
89      private boolean isOnGrass(int x, int y){
90        // Checks if a coordinate is on grass
91          if(x == 0)
92            return true;
93          if(x == this.rows)
94            return true;
95          if(y == 0)
96              return true;
97          if(y == this.columns)
98              return true;
99          return false;
100     }
101 }
```

Y la vista se implementa de la siguiente manera:

```
1  package Environment;
2
3  import jason.environment.grid.GridWorldView;
4
5  import java.awt.Color;
6  import java.awt.Font;
7  import java.awt.Graphics;
8  import java.util.logging.Logger;
9
10
11 public class PainterView extends GridWorldView{
12
13   private static final long serialVersionUID = 1L;
14   Logger logger = Logger.getLogger(PainterView.class.getName());
15
16   PainterModel model;
17
18   public PainterView(PainterModel model) {
19     super(model, "Painter", 700);
20     this.model = model;
21     defaultFont = new Font("Arial", Font.BOLD, 16);
22     setVisible(true);
23     repaint();
24   }
25
26     @Override
27     public void draw(Graphics g, int x, int y, int object){
28       switch (object){
29       // Red paint
30       case 32:
31         g.setColor(Color.red);
32             g.fillRect(x * cellSizeW-1, y * cellSizeH-1, cellSizeW-1, cellSizeH-
                   ↪ 1);
33             break;
34         // Add grass
35       case 16:
36         g.setColor(Color.green);
37             g.fillRect(x * cellSizeW-1, y * cellSizeH-1, cellSizeW-1, cellSizeH-
                   ↪ 1);
38         break;
39       }
40     }
```

```
41
42      public void drawAgent(Graphics g, int x, int y, Color c, int id){
43        super.drawAgent(g, x, y, Color.gray, -1);
44        g.setColor(Color.black);
45        drawString(g, x, y, defaultFont, "Painter");
46      }
47  }
```