Primeros pasos con PLEXIL and the Universal Executive

Pablo Muñoz Martínez
Juan Alejandro Mora Prieto
Departamento de Automática
Universidad de Alcalá

Primeros pasos con PLEXIL and the Universal Executive
Última actualización: 9 de Diciembre de 2013.

Si ha encontrado algún error en el documento o desea contactar con el autor, puede hacerlo a través de la siguiente dirección de email: `pmunoz@aut.uah.es`. Todos los comentarios serán bien recibidos.

Índice general

Índice de figuras

Capítulo 1

Introducción

Este documento representa una guía básica de uso y funcionamiento del lenguaje PLEXIL (*Plan Execution Interchange Language*) para modelado de planes de alto nivel enfocados a sistemas autónomos y del ejecutor de planes diseñado para él: el *Universal Executive* (en adelante UE).

PLEXIL fue originalmente desarrollado como un proyecto colaborativo entre investigadores de la NASA y de la Universidad Carnegie Mellon, bajo fondos del *Mars Technology Program* a través del *Research Institute for Advanced Computer Science* (RIACS) en la *Universities Space Research Association* (USRA). El proyecto comenzó en 2006 y actualmente esta en constante evolución. Se encuentra bajo licencia BSD (*Berkeley Software Distribution*) y se puede descargar el código fuente completo con ficheros de ejemplo desde la página del proyecto en SourceForge [1]. También puede consultarse en SourceForge la wiki del proyecto accediendo al enlace a través de la página de descarga del mismo.

PLEXIL y el UE han sido utilizados para demostración integrándolo en el *rover* K10 (figura 1) para pruebas de movilidad y de interacción hombre-robot. El K10 está preparado para realizar tareas de inspección. El programa desarrollado en el *Ames Research Center* le permite realizar una inspección completa de 360º a un SCOUT *rover* tomando fotografías de alta resolución en una serie de puntos predeterminados [2]. También ha sido probado en el *Mars Drill*, un prototipo de taladro gestionado de forma autónoma. Finalmente, se ha empleado para demostrar la automatización de algunas tareas en la Estación Espacial Internacional (ISS).

Figura 1.1: Robot K10

En éste primer capítulo se dará una descripción breve del lenguaje PLEXIL y el UE, así como los elementos necesarios para poder utilizarlos y la instalación de los mismos. En el segundo capítulo se abordará el lenguaje PLEXIL, y en el tercero el funcionamiento del UE.

1.1. Requisitos

Las necesidades de los programas auxiliares de PLEXIL y el UE vienen descritas en el manual de PLEXIL [3] (este manual viene incluido en la distribución de PLEXIL), habiendo sido probado correctamente en Red Hat Linux (versiones 4 y 5), MacOS X (10.4, 10.5 y 10.6) y Ubuntu[1] (desde la versión 8.04 hasta la 10.04), pero debería funcionar en la mayoría de los sistemas basados en UNIX.

Para poder almacenar el código fuente y los ejecutables es necesario disponer de al menos 150MB de espacio en disco. El uso de memoria RAM no está calculado y depende de varios factores, no obstante, está diseñado con un enfoque eficiente para poder utilizarlo bajo sistemas empotrados con bajos recursos.

También será necesario disponer de varios programas para poder compilar el código y generar los correspondientes ejecutables para PLEXIL, el UE y otras herramientas de utilidad incluidas en la distribución:

- **g++**: el compilador de C++, versión 3.3.3 o superiores.
  (http://gcc.gnu.org)
- **ant**: herramienta para la automatización de la compilación.
  (http://ant.apache.org/)
- **Java**: tanto la máquina virtal de Java como el JDK (*Java Development Kit*). Requiere la versión 1.5 o 1.6 (no se recomienda el uso de OpenJDK por carecer de un jar requerido).
  (http://www.java.com)
- **Doxygen**: herramienta opcional que permite generar automáticamente la documentación del código en varios formatos.
  (http://www.stack.nl/~dimitri/doxygen)

1.2. Instalación

Una vez descargada la distribución de plexil, habrá que realizar la compilación del código fuente. Para ello, primero habrá que descomprimir los archivos en una carpeta, en este caso usaremos `/home/usuario/plexil` como origen y después será necesario definir la variable PLEXIL_HOME y añadir una nueva ruta al PATH de la `shell`. Será por tanto necesario modificar el archivo de inicio de la `shell` que utilicemos en nuestro sistema. Por ejemplo, para `bash` este archivo se ubicará en `/home/usuario/.bashrc`. Para incluir las rutas necesarias será preciso insertar las líneas que se indican en las figuras 1.2 ó 1.3 en el archivo correspondiente en función del sistema usado.

```
1 setenv PLEXIL_HOME /home/usuario/plexil
2 setenv PATH $PLEXIL_HOME/bin:$PATH
3 (No Mac) setenv LD_LIBRARY_PATH $PLEXIL_HOME/lib
4 (sólo Mac) setenv DYLD_LIBRARY_PATH $PLEXIL_HOME/lib
5 (sólo Mac) setenv DYLD_BIND_AT_LAUNCH YES
```

Figura 1.2: Variables de entorno para `csh`

[1] Para poder compilar `robosim` se deben tener instalados los siguientes paquetes: `glutg3`, `glutg3-dev`, `libxi-dev` y `libxmu-dev`

```
export PLEXIL_HOME=/home/usuario/plexil
export PATH=$PLEXIL_HOME/bin:$PATH
(No Mac) export LD_LIBRARY_PATH=$PLEXIL_HOME/lib
(sólo Mac) export DYLD_LIBRARY_PATH=$PLEXIL_HOME/lib
(sólo Mac) export DYLD_BIND_AT_LAUNCH=YES
```

Figura 1.3: Variables de entorno para `bash`

Una vez definidas las variables en el archivo de configuración de la `shell` se podrán utiliar los ejecutables de PLEXIL y el UE desde cualquier directorio de forma cómoda, además de poder emplear las librerías dinámicas de la distribución en nuestros propios programas. Con esto realizado, habrá que situarse sobre el directorio de PLEXIL y ejecutar `make` para que de forma automática compile y construya todos los elementos del sistema. El archivo `Makefile` sólo ejecuta los correspondientes comandos que compilan cada elemento. Si se quiere realizar sólo la compilación de un elemento, basta con elegir la orden correspondiente.

Cada objetivo del archivo construye un elemento[2]. `UniversalExec` construye el ejecutable del UE (capítulo 3), `luv` el visor gráfico LUV (vease la sección 3.2.2), `standard-plexil` prepara el sistema de parseado y traducción para los planes de PLEXIL (más detalles en el capítulo 2), y el resto de líneas permiten generar herramienas auxiliares o simuladores de ejemplo incluidos en la distribución.

La forma de ejecutar cada uno de los ejecutables generados se verá posteriormente en sus respectivas secciones.

Una vez realizada la construcción de los elementos, se puede realizar la generación de documentación del código. Para ello, basta con ejecutar las ordenes mostradas en la figura 1.4. Con ello obtendremos la documentación del código en `$PLEXIL_HOME/src/exec/doc/index.html`.

```
cd $PLEXIL_HOME/src/exec
doxygen ue.dox
$
```

Figura 1.4: Generación de la documentación con Doxygen

1.2.1. Control de subversión

PLEXIL está mantenido en un sistema de control de subversiones (SVN). Mediante dicho sistema, tras la instalación inicial de PLEXIL el usuario podrá interactuar con el SVN y actualizar los archivos de la distribución a las últimas versiones. Para poder realizar las acciones que se detallan a continuación, habrá que disponer del programa SVN en la máquina local. Para obtener el programa puede dirigirse a su página oficial:
`http://subversion.tigris.org/`.

El script `plexil/bin/checksvn` permitirá conocer si hay alguna versión nueva de los archivos en el repositorio SVN. Si se desea actualizar u obtener la última versión disponible, basta con introducir la orden de la figura 1.5.

Con ello obtendremos una copia actualizada (o actualizaremos la copia local) de todos los componentes de PLEXIL. La copia se albergará en una carpeta de nombre

[2]Durante la compilación es posible que se den avisos por el uso de funciones obsoletas.

```
1 svn checkout https://plexil.svn.sourceforge.net/svnroot/
    plexil/trunk plexil
```

Figura 1.5: Obtención de la última versión de PLEXIL

`plexil` en el directorio desde el cual ejecutemos la orden. Si desea conocer más acerca de SVN puede consultar [4].

Capítulo 2

El lenguaje PLEXIL

En este capítulo se verán los elementos que conforman el lenguaje de ejecución PLEXIL y la codificación de los mismos, además de como llevar a cabo la traducción de los planes escritos en las diferentes versiones del lenguaje PLEXIL a XML mediante las herramientas desarrolladas a tal efecto. Además se verá la codificación de PlexilScript, con el cual podremos realizar simulaciones de planes mediante la aplicación TestExec que se verá en el capítulo 3.

2.1. Introducción a PLEXIL

PLEXIL es un lenguaje para el modelado y representación de planes de ejecución de sistemas autónomos. Es válido tanto para sistemas mono-agente como para multi-agente ya sea en entornos reales o simulados. Está diseñado de tal forma que sea compacto, semánticamente sencillo y determinista siempre que se de la misma secuencia de sucesos en el mundo exterior. A pesar de ser un lenguaje sencillo, permite definir bucles, condiciones de salto, actividades basadas en tiempo y eventos, actividades concurrentes o en secuencia y permite restricciones temporales y manejo de recursos. El núcleo de la sintaxis es simple y uniforme, lo que permite una fácil y rápida interpretación de los planes escritos por parte del planificador.

Hay que destacar que PLEXIL, al contrario que otros lenguajes de planificación como, por ejemplo, PDDL [5], tiene una única entrada, es decir, el plan y el problema son un único elemento. Esto es así ya que un plan de PLEXIL consta de una serie de elementos de acción que podrán ser o no ejecutados en función del estado del mundo, la ejecución correcta o no de otras acciones y, a su vez, estas acciones constituyen metas que deben ser alcanzadas. El plan finalizará una vez todas las posibles acciones que se puedan ejecutar, hayan sido ejecutadas (independientemente del resultado de la ejecución). Estos elementos de ejecución son los nodos.

El formato de lenguaje PLEXIL será XML, aunque la codificación de los planes podrá realizarse de dos formas: utilizando el lenguaje PLEXIL estándar (en adelante Plexil) o PlexilLisp. La codificación que se verá en las siguientes secciones corresponde al lenguaje PLEXIL estándar. Con estos planes escritos, mediante un programa creado para ello, se realizará el análisis sintáctico y semántico del mismo y se encargará de realizar la traducción al formato XML. El archivo XML generado será el que utilizará el ejecutor como elemento de entrada.

2.2. Nodos

Los nodos son el elemento fundamental del lenguaje PLEXIL. Todo plan escrito en PLEXIL constará de uno o más nodos. Cada nodo a su vez constará de dos elementos:

- **Condiciones**: una serie de condiciones que permitirán controlar cuando puede ejecutarse el cuerpo del nodo, que ha de ocurrir para que finalice correctamente o cuando puede volver a ser apto para ejecución por ejemplo. Todas las posibles condiciones se verán en la sección 2.4.
- **Cuerpo**: el cuerpo del nodo determina las acciones que llevará a cabo el nodo. En función de la acción que ejecute el nodo se considerará un tipo u otro de nodo. Por ejemplo, existen nodos de asignación, de comando o de llamada a función. Los tipos de nodos y sus particularidades se verán en la sección 2.5.

Un plan de PLEXIL será un árbol de nodos que representará una descomposición jerárquica de tareas que debe llevar a cabo el plan. En este sentido, lo normal será distribuir las tareas de tal forma que las de más alto nivel se sitúen cercanas a la raíz del árbol y las de menor nivel se aproximen a las hojas. El número de ramas y la profundidad del árbol la determinaremos a la hora de escribir el plan en función de nuestras necesidades. En la figura 2.1 puede verse gráficamente un plan de ejemplo.

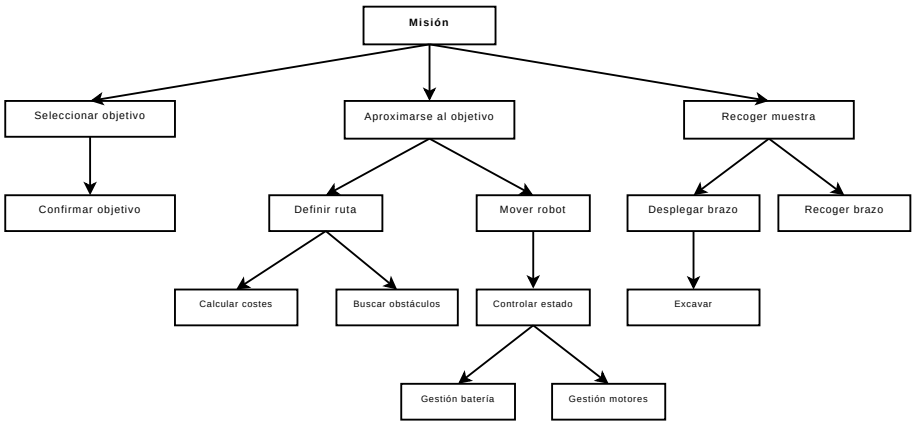

Figura 2.1: Ejemplo gráfico de un plan escrito en PLEXIL

En las siguientes subsecciones veremos como se declara un nodo y los atributos que pueden utilizarse dentro del lenguaje. Tras ello, se analizarán las siete condiciones posibles que se pueden utilizar para controlar la ejecución de los nodos y los seis tipos de nodos disponibles en PLEXIL.

2.2.1. Declaración

Para declarar un nodo en lenguaje Plexil se recurre a la estructura mostrada en la figura 2.2. En la declaración, *nombre* representa la identificación unívoca del nodo. Los datos del nodo serán variables, condiciones de ejecución y los datos propios del nodo en función del tipo de nodo que sea (las variables y condiciones son opcionales). El tipo de nodo no es necesario declararlo, ya que será determinado automáticamente en función de los datos que se introduzcan en él.

Un nodo puede no estar nombrado, en cuyo caso será únicamente un par de corchetes en cuyo interior estará el cuerpo del nodo. Aunque esto es válido, no suele ser práctico ya que no puede referenciarse de ninguna manera al nodo.

```
nombre:
{
   <datos>
}
```

Figura 2.2: Declaración de un nodo en Plexil

2.3. Variables

En Plexil se pueden declarar variables locales dentro de un nodo. Existen cuatro tipo de variables: `Boolean, Integer, Real` y `String`. Además se podrán declarar vectores para cada tipo simple. Hay que indicar que sólo existen variables, no se pueden declarar constantes como tal.

Los variables se declaran como sigue: `tipo nombre = valor;` donde la asignación inicial puede omitirse, tomando entonces `UNKNOWN` como valor la variable declarada. En la figura 2.3 pueden verse varios ejemplos de declaración de variables.

```
Boolean iniciar = true;
Integer pasos;
Real pi = 3.141592;
String mensaje = "Inicializado";
```

Figura 2.3: Ejemplos de declaración de variables

Como se ha indicado, para cada uno de los cuatro tipos simples, se pueden declarar vectores o arrays unidimensionales. Se declaran de la siguiente forma: `tipo nombre[dimensión] = #(val1 val2 ...);`, debiendo siempre especificarse la dimensión del mismo. Para trabajar con los arrays hay que trabajar con cada elemento independientemente referenciado por su posición (`array[pos]`), no pudiendo trabajar con el array completo. En la figura 2.4 pueden verse ejemplos de declaración de arrays. Los arrays al igual que las variables simples se inicializarán a `UNKNOWN` en caso de no especificar los valores, salvo que en el caso de los arrays pueden especificarse los primeros n valores, dejando el resto indeterminados. Es importante destacar que los arrays comienzan en la posición 0.

```
Integer temperaturas[100];
Real predefinido[10] = #(1.3 2.0 3.5);
Integer X[6] = [1,3,5];
```

Figura 2.4: Ejemplos de declaración de arrays

2.3.1. Visibilidad de variables

Las variables serán visibles localmente por el nodo que las haya declarado en su cuerpo y por todos sus hijos. Este es el comportamiento general y se puede modificar explicitamente mediante el empleo de clausulas de interfaz. Estas clausulas definirán los variables que serán accesibles a los hijos del nodo que las declara y la forma en que estos pueden acceder. Si se ha declarado la interfaz para al menos una variable en el nodo, el resto de variables cuya interfaz no sea declarada se asumirá directamente que son inaccesibles a los hijos. Las dos clausulas de interfaz son las siguientes:

- `In`: esta clausula permite que los hijos lean las variables. Una variable de lectura no podrá ser redeclarada por un hijo como lectura y escritura.
- `InOut`: permite la lectura y escritura de la variable por los nodos hijos.

```
1 Interfaz:
2 {
3   In x, y;
4   InOut z;
5   Integer a, b;
6 }
```

Figura 2.5: Ejemplo de interfaces

En la figura 2.5 puede verse un ejemplo de un nodo con una serie de variables. En dicho nodo, las variables x e y serán accesibles a sus descendientes como variables de sólo lectura, mientras que z lo será para lectura y escritura. Finalmente, las variables a y b, al no tener una interfaz declarada, no serán accesibles por los nodos hijos.

2.3.2. Expresiones

A la hora de actuar con variables se recurre a las expresiones. La forma de estas expresiones dependerá del tipo de variables sobre el que trabajen, pero una expresión en PLEXIL será esencialmente un valor literal, una variable, una consulta al estado del mundo exterior, el valor de un estado de un nodo, o una combinación de estos elementos. Por tanto, una expresión puede contener expresiones combinadas entre ella mediante el uso de operadores.

Además de aparecer para la asignación de variables, las expresiones podrán aparecer en las condiciones que gestionan la ejecución de los nodos o en la gestión de recursos.

Para el caso de operaciones numéricas, se podrá recurrir a los operadores tradicionales: suma, resta, multiplicación y división, así como a raíces cuadradas[1] (`sqrt`) y valor absoluto[2] (`abs`). Será posible en algunos casos mezclar números reales y números enteros siendo el tipo resultante dependiente del contexto[3]. Las reglas de precedencia y asociación de operadores son las reglas estándar, permitiéndose el uso de paréntesis para especificar el orden de operación. En la figura 2.6 se pueden ver ejemplos de expresiones numéricas.

```
1 resultado = numreal / 14.5;
2 x1 = -b + (sqrt(b*b - 4*a*c) / (2*a));
3 tempVal = (temperatura[2] - 32) * 5 / 9;
4 absoluto = abs(real);
```

Figura 2.6: Ejemplos de expresiones numéricas

Las expresiones lógicas vendrán determinadas por los operadores de comparación (igualdad: `==`, desigualdad: `!=`, mayor que o mayor o igual que: `> o >=` y menor que

[1] Si se intenta obtener la raíz de un número negativo (resultado complejo) se producirá un error en la ejecución.

[2] Aunque el valor absoluto está definido, no es así para el negativo, el signo '-' sólo es válido como resta, no se permite como inversor de signo. Para obtener el negativo a partir de un número positivo, se deberá recurrir a la expresión (0-x).

[3] Estos casos no están documentados, aunque son intuitivos.

o menor o igual que: `< o <=`) o los operadores lógicos (negación: `!`, disyunción u OR: ||, conjunción o AND: `&&` y disyunción exclusiva o XOR: `XOR`). Con estos operadores se podrán formar expresiones cuyo resultado será siempre un valor lógico (consulte la sección 2.3.3 para conocer los valores lógicos posibles). En la figura 2.7 pueden verse ejemplos de expresiones lógicas.

```
1 enmovimiento = velocidad > 0;
2 estado = reconocido && !(contador <= 30);
3 conocido = isKnown(valor);
```

Figura 2.7: Ejemplos de expresiones lógicas

Para las cadenas de caracteres se permite sólo el paso de éstas como parámetros. la concatenación (`+`) y la comparación de igualdad (`==`) o desigualdad (`!=`) entre cadenas. En la figura 2.8 se pueden ver ejemplos de operación con cadenas de caracteres.

```
1 mensaje = "Bienvenido ";
2 entrada = mensaje + "usuario";
```

Figura 2.8: Ejemplos de expresiones con cadenas

Finalmente, se podrá operar con los elementos del array, y nunca con el array completo. Para operar con un elemento del array habrá que extraerlo mediante el operador `[i]`, donde `i` es el índice, comprendido entre 0 y el tamaño máximo del array. Para asignar el valor a un elemento del array se referenciará al elemento de la misma manera. Además es posible asignar un array a otro, de tal manera que si el array de destino es mayor que el origen, los elementos restantes obtendrán el valor `UNKNOWN`. En caso de que el array destino fuera menor que el origen se produciría un error. En la figura 2.9 se puden ver varios ejemplos de manipulación de arrays.

```
1 Integer numeros[9];
2 Integer X[6] = [1,3,5];
3 numeros[8] = 9;
4 X[5] = numeros[8] * X[2];
5 numeros = X;
```

Figura 2.9: Ejemplos de expresiones con arrays

A continuación se describen varias operaciones que no admiten el uso de arrays:

- Un array como valor de retorno de un comando o llamada a función.
- Arrays completos como parte de una expresión (salvo asignación a otro array).
- Arrays enteros como parámetro de una función (sólo se permiten elementos).
- Arrays como variables dentro de una interfaz.

PLEXIL implementa un tipo de expresiones para el tiempo y la duración (*Date and Duration expressions*) que son definidas bajo el estándar ISO-8601 [6]. Este tipo de expresiones están pensadas para el uso de operaciones temporales para *lookups* y actividades que requieran referencias de tiempo real. Permiten a su vez el uso de operaciones aritméticas como las vistas anteriormente.

```
1  // Tiempo UTC
2  Date fecha1 = Date("2013-12-12T09:43:00.00Z");
3
4  // Hora local
5  Date fecha2 = "2013-15-03T20:32:00.00";
6
7  // Duración de 45 minutos
8  Duration dur1 = Duration("PT45M");
9
10 // Ejemplo de operaciones
11 Date fecha3;
12 Duration dur2;
13
14 // Resta de 3.2 segundos a la fecha1
15 fecha3 = fecha1 - Duration("PT3.2S");
16
17 // Resta de 2 minutos a la fecha2
18 dur2 = fecha2 - Date("2013-15-03T20:30:00.00");
```

Figura 2.10: Ejemplos de expresiones de tiempo y duración

2.3.3. Lógica trivaluada y el valor `UNKNOWN`

En PLEXIL se utiliza una lógica de tres valores, en la cual a los valores típicos de las lógicas bivaluadas (`true` y `false`) se le añade el valor `UNKNOWN`. Por tanto, los operadores lógicos se han redefinido para operar con este valor como se ve en la figura 2.11.

```
1  True    && UNKNOWN = UNKNOWN
2  False   && UNKNOWN = False
3  True    || UNKNOWN = True
4  False   || UNKNOWN = UNKNOWN
5  UNKNOWN && UNKNOWN = UNKNOWN
6  UNKNOWN || UNKNOWN = UNKNOWN
7           ! UNKNOWN = UNKNOWN
```

Figura 2.11: Operadores lógicos y el valor `UNKNOWN`

Además, como ya se indicó, el valor `UNKNOWN` puede aparecer como valor de cualquier tipo de variable, y, por tanto, puede ser resultado de una expresión. Esto último ocurrirá en los siguientes casos:

- Aparece como valor inicial de una variable (ya sea simple o elementos de un array) por no haber declarado el valor inicial.
- Es el estado final inicial de un nodo.
- Como resultado de una consulta al estado del mundo fallida.
- Como resultado de consultar acerca de un evento temporal que no ha ocurrido.

Este valor no aparece de forma literal, es decir, no se puede asignar de forma directa y solo aparece por las condiciones expuestas anteriormente. No obstante, mediante el empleo de la función `isKnown(`*`variable`*`)` se puede comprobar este valor. Si la función retorna `false`, entonces el argumento se puede evaluar a `UNKNOWN` y, en caso contrario, retornará `true`.

2.4. Condiciones

Para el control de la ejecución de cada nodo al inicio, fin o durante, se pueden definir hasta siete condiciones. Cada condición irá seguida de una expresión booleana: `tipo_condicion <expresión_booleana>;`. Estas siete condiciones se enmarcarán en dos grupos: las *gate conditions*, que comprende las condiciones de inicio, finalización, repetición y omisión, y el grupo denominado *check conditions*, dentro del cual están las precondiciones, postcondiciones y condiciones invariables. El primer grupo determinará cuando el nodo puede comenzar o finalizar su ejecución y el segundo grupo permite determinar condiciones por las cuales la ejecución del nodo no fue correcta. En la figura 2.12 puede verse un ejemplo de un nodo con todas las condiciones definidas.

A la hora de trabajar con las condiciones será muy útil (y necesario en muchos casos) permitir o no la ejecución de un nodo en función del resultado de la ejecución de otro. Para ello se podrán utilizar los estados de los nodos (veáse la sección 2.9) en las condiciones.

```
1  subir_temperatura:
2  {
3      StartCondition temperaturaBaja;
4      EndCondition temperatura >= 20 || isKnow(temperatura);
5      RepeatCondition temperatura < 20;
6      SkipCondition temperatura > 20 || !temperaturaBaja;
7      PreCondition temperatura > -10 && temperatura <30;
8      PostCondition temperatura > -10;
9      InvariantCondition !temperaturaBaja;
10     Command: temperatura = SubirTemp(incremento);
11 }
```

Figura 2.12: Ejemplo de nodo con todas las condiciones

2.4.1. Condición de inicio, *StartCondition*

Esta condición indica cuando el nodo puede comenzar a ejecutarse. En el momento que la condición indicada se cumpla, éste comenzará su ejecución. En caso de no haber, el nodo iniciará la ejecución en el momento que la ejecución del plan alcance el nivel de profundidad del nodo en cuestión.

2.4.2. Condición de finalización, *EndCondition*

Indica la condición para que el nodo termine su ejecución. En caso de omitirse, el nodo terminará la ejecución una vez haya realizado la acción de su cuerpo.

2.4.3. Condición de repetición, *RepeatCondition*

Esta condición permite que un nodo que ya ha sido ejecutado vuelva a ser apto para ejecutarse. Para ello, el nodo en cuestión deberá cumplir con la condición de repetición y la condición de inicio y precondición en caso de tenerlas. Hay que tener en cuenta que un nodo que ya ha sido ejecutado (correcta o incorrectamente) no se volverá a ejecutar salvo que se indique una condición de repetición.

2.4.4. Condición de omisión, *SkipCondition*

La condición de omisión permite establecer un nodo como ejecutado por omisión (que implica que el nodo haya finalizado su ejecución). En caso de que un nodo posea esta condición y se verifique en el momento en que pueda entrar en ejecución, el nodo no será ejecutado. Esta condición permite definir saltos del tipo `if-then-else` combinándola con otras condiciones.

2.4.5. Precondición, *PreCondition*

Esta condición se evalúa después de la condición de inicio y determina si es seguro proceder a la ejecución del nodo. Para que un nodo pueda ejecutarse, deberá verificar primero la condición de inicio y luego la precondición. En caso de no cumplir la precondición, el nodo no podrá ejecutarse y será abortado indicando que la ejecución ha sido fallida.

2.4.6. Postcondición, *PostCondition*

La postcondición se evalúa tras la ejecución del nodos y después de evaluar la condición de finalización. Esta condición permite comprobar si el nodo ha realizado su función correctamente. Si la condición especificada no se cumple, el nodo no habrá cumplido con sus objetivos y se marcará como fallido.

2.4.7. Condición invariable, *InvariantCondition*

Esta condición será comprobada durante toda la ejecución del nodo de tal forma que deberá cumplirse en todo momento. Si en el transcurso de la ejecución se da alguna circunstancia que implique que la condición invariable no se cumpla, se abortará la ejecución del nodo y se marcará como fallido. Es una condición útil para determinar las condiciones de seguridad en las cuales debe ejecutarse el nodo.

2.4.8. Jerarquía de condiciones

Las condiciones se pueden agrupar en una jerarquía en base al control de inicio, ejecución y finalización del nodo. Esta jerarquía puede verse en la figura 2.13 y es como sigue:

- **Inicio del nodo**: la primera condición que se verificará es la condición de omisión, sólo si ésta se verifica podrán comprobarse el resto, sino se omitirá el nodo. Después se comprobará la condición de inicio, no comprobándose el resto hasta que esta no se satisfaga. Una vez comprobada esta condición, se podrá ver si se cumple la precondición, y, en caso de cumplirse, se ejecutará el nodo. Si la precondición no se cumple, el nodo finalizará por fallo en la precondición. En caso de que el nodo disponga de una condición de repetición, será la última en comprobarse siempre que el nodo ya se haya ejecutado al menos una vez.

- **Ejecución del nodo**: durante la ejecución del nodo se comprobará la condición invariante. Está deberá verificarse durante todo el período de ejecución del nodo, provocando el aborto de la ejecución en caso de no cumplirse. En dicho caso, lo indicará mediante un fallo en la condición invariante.

- **Finalización del nodo**: una vez ejecutado el nodo, se comprobará la condición de finalización. En caso de cumplirse, será comprobada la postcondición. Hasta que no se verifique la condición de finalización el nodo seguirá en estado de ejecución. Si la condición de finalización es correcta, pero la postcondición

no, el nodo finalizará con un fallo por la postcondición.

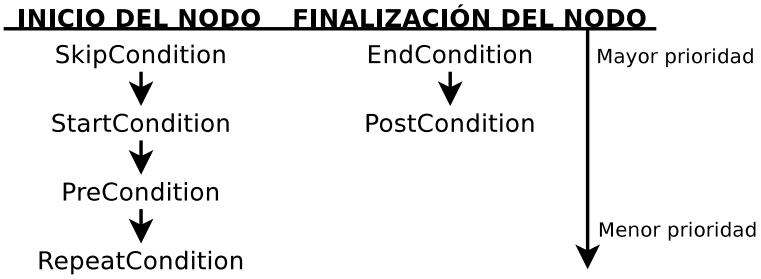

Figura 2.13: Jerarquía de condiciones

2.5. Tipos de nodos

En esta sección se describirán los seis[4] tipos de nodos existentes en PLEXIL. Como ya se dijo anteriormente, los nodos se estructuran como un árbol. En este sentido, se conservarán las definiciones clásicas: el nodo superior en la jerarquía será la raíz, siendo un nodo *lista* del cual colgarán más nodos que pueden ser hojas (nodos de cualquier tipo que no sea lista) u otro nodo lista que generará otro nivel de profunidad en el árbol. Será importante la distinción entre nodo padre e hijo: un nodo padre será un nodo lista y los hijos los nodos inmediatamente inferiores que dependan de dicho nodo. Por tanto, un nodo lista será siempre un nodo interior y cualquier otro tipo de nodo será un nodo hoja e hijo de un nodo lista. Serán estos nodos hojas los encargados de interactuar con el mundo y realizar las acciones pertientes del plan.

A continuación se analizarán los seis tipos de nodos. Aquí se verán los elementos que conforman parte del cuerpo del nodo para cada tipo particular.

2.5.1. Nodo vacío, *Empty node*

Un nodo vacío es aquel que no realiza ninguna acción. Estos nodos sólo pueden contener atributos y en la práctica son poco comunes y su utilidad suele ser verificar el estado del mundo exterior. En este sentido tendrá alguna condición que permita realizar una comprobación y, en base a ello, generar un estado de salida del nodo que sea útil para la ejecución de otro nodo.

```
1 ExcesoTemp:
2 {
3     PostCondition LookupNow("temperatura") > 100.0;
4 }
```

Figura 2.14: Ejemplo de nodo vacío

[4]Existe un séptimo tipo de nodo, el nodo de solicitud de planificación (*Plan Request node*), que está diseñado y parcialmente implementado. Este nodo soporta interacción con planificadores externos permitiendo un punto de enganche para añadir un nuevo plan. En este momento este tipo no esta soportado oficialmente por PLEXIL ni es plenamente funcional.

2.5.2. Nodo de asignación, *Assignment node*

Un nodo de asignación permite modificar el estado de una variable mediante una expresión lógica o matemática. En el lado izquierdo de la expresión deberá haber una variable apta para escritura y en el lado derecho una expresión. Esta expresión constará de, al menos, un operando o literal (en caso de una asignación simple) o dos operandos unidos por un operador. En todos los casos, el resultado de la expresión debe ser del mismo tipo que la variable a la cual se asignará el resultado. Los operandos pueden ser una varible simple, un elemento de un array (nunca un array completo) o un literal. Dentro de la expresión se pueden utilizar paréntesis para priorizar los cálculos. La clausula de asignación y un ejemplo de nodo de asignación se pueden ver en la figura 2.15. Sólo podrá haber una clausula de asignación por nodo con una única asignación. En actuales revisiones de PLEXIL, se permite la omisión del *tag* `Assignment:`.

```
1  Assignment: <variable> = <expresion>;
2
3  incrementarAngulo:
4  {
5      Assignment: angulo = angulo + 25;
6  }
7
8  convertirTempSens2
9  {
10     Assignment: temperatura = (sensorTemp[2] - 32) * 5 / 9;
11 }
```

Figura 2.15: Clausula de asignación y ejemplos

2.5.3. Nodo de comando, *Command node*

Los nodos de comando permiten ejecutar acciones soportadas e implementadas en el sistema a controlar. El funcionamiento de estos nodos será llamar a una función externa con los parámetros adecuados. Estos parámetros será una lista formada por literales o variables, pero nunca por expresiones. Además, si la función implementada retorna un valor de un tipo reconocido por PLEXIL, podrá ser asignado a una variable. En la figura 2.16 puede verse la clausula que determina al nodo de comando. Ésta especifica el nombre del comando a ejecutar seguido por la lista de parámetros que utilizará entre paréntesis. La lista de parámetros estará separada por comas y será opcional. En caso de querer asignar el valor resultante a una variable, bastará con indicarlo mediante la asignación correspondiente, siempre y cuando sea posible escribir en dicha variable desde el nodo. En actuales revisiones de PLEXIL, se permite la omisión del *tag* `Command:`.

Los nodos de comando no bloquean la ejecución, es decir, la ejecución de los comandos se lleva a cabo de manera asíncrona. A todo comando en ejecución se le asigna un manejador para que, una vez finalice, lo indique de forma inmediata al ejecutor. En caso de que el comando retorne un valor, y ya que la ejecución es asíncrona, es común definir una condición de finalización del nodo para que éste sólo finalice una vez recibido el dato. Esto puede hacerse mediante la función `isKnown()`, ya que durante la ejecución del comando el valor de retorno se establece a `UNKNOWN`.

Además, se pueden establecer necesidades y usos de recursos para la ejecución de comandos; ésto se verá en la sección 2.11.

```
Command: [<variable> =] <comando> [(<lista_args>)];

Rotar:
{
   StartCondition angulo != 0;
   RepeatCondition angulo != 0;
   Command: rotar(angulo);
}

ConfirmarOrden:
{
   Boolean resultado;
   EndCondition isKnown(resultado);
   PostCondition resultado;
   Command: resultado = SolConfirm("¿Ejecutar instrucción?",
       numInstruccion);
}
```

Figura 2.16: Clausula de comando y ejemplos

2.5.4. Nodo de actualización, *Update node*

Este nodo permite actualizar el valor de una o más variables en un único nodo, sustituyendo el valor actual de la variable por el valor de otra variable o de un literal, pero nunca por el resultado de una expresión. Para ello, hay que declarar la clausula `Update` seguida de una lista separada por comas y formada por pares `variable=variable` o `variable=literal`, que representarán la variable que queremos actualizar a la izquierda y el nuevo valor a la derecha. En la figura 2.17 puede verse un ejemplo de nodo de actualización.

```
Update: (<nombre> = (<valor> | <variable>))*;

ActualizarDatos:
{
   Integer actualizado;
   Update: actualizado = resultado[1], anterior = -2;
}
```

Figura 2.17: Clausula de actualización y ejemplo

2.5.5. Nodo de llamada a librería, *Library call node*

Estos nodos permiten enlazar con nodos librería, cuyos detalles se verán en la sección 2.7. Los nodos de llamada a librería constan de una clausula `LibraryCall` tal y como se muestra en la figura 2.18. En ella puede verse el identificador del nodo librería que será utilizado, así como la posibilidad de renombrarlo en el contexto del plan en ejecución si así se desea. Finalmente, aparece una lista de argumentos que actuarán como si de una función se tratará. Está lista deberá declarar el nombre del parámetro y mediante una asignación indicar el valor que éste debe tomar al realizar la llamada. Sólo será posible asignar literales o variables. La codificación completa junto a un ejemplo se verá en la sección 2.7.

```
1  LibraryCall: <IdNodo> [<NuevoIdNodo>] [<lista_alias>];
```

Figura 2.18: Clausula de llamada a librería

2.5.6. Nodo lista, *List node*

Un nodo lista es un nodo que permite declarar una lista de cero o más nodos hijos. Estos nodos son los que le dan la estructura jerárquica a PLEXIL. Los hijos de un nodo lista podrán ser de cualquier tipo de nodo, incluyendo nodos lista. Para declarar un nodo lista basta con especificar la clausula `NodeList` seguida de la declaración de los nodos hijos deseados como puede verse en la figura 2.19.

```
1  NodeList:
2  <nodo1>
3  ...
4  <nodoN>
5
6  root:
7  {
8     Integer x;
9     NodeList:
10    Informar:
11    {
12       Command: Mostrar("Plan en ejecución...");
13    }
14    LeerX:
15    {
16       Command: x = GetX();
17    }
18    Accion:
19    {
20       Boolean resultado;
21       StartCondition LeerX.state == FINISHED;
22       EndCondition isKnown(resultado);
23       PostCondition resultado;
24       Command: resultado = Operar(x);
25    }
26    InformarCorrecto:
27    {
28       StartCondition Accion.outcome == SUCCESS;
29       SkipCondition Accion.outcome == FAILURE;
30       Command: Mostrar("Operación correcta");
31    }
32    InformarFallo:
33    {
34       StartCondition Accion.outcome == FAILURE;
35       SkipCondition Accion.outcome == SUCCESS;
36       Command: Mostrar("Operación fallida");
37    }
38 }
```

Figura 2.19: Clausula de lista y ejemplo

Los nodos lista ejecutarán por defecto todos los nodos hijos de forma concurrente y, la ejecución del nodo finalizará una vez todos los nodos hijos hayan finalizado su ejecución. Si se desea controlar la secuencia de ejecución de los nodos habrá que recurrir a la utilización de condiciones de ejecución. Gracias a esto, se puede realizar

la ejecución de nodos en serie siguiendo la pauta deseada, realizar bucles en función de los resultados, o saltar determinados nodos en función de la ejecución de otros. En este último caso, dado que el nodo padre sólo finalizará tras la ejecución de los hijos, será necesario incluir una condición para omitir la ejecución del nodo cuando no se den las condiciones necesarias para su ejecución, de cara a que el nodo padre pueda finalizar su ejecución y no haya peligro de provocar un bloqueo en otros nodos en el nivel del padre.

2.6. Declaraciones globales

Para los planes que posean llamadas a funciones, comandos, nodos librerías o consultas al mundo exterior, es recomendable (y obligatorio para los nodos librería) definir la interfaz de las mismas usando declaraciones globales. Estas declaraciones será una lista separada por ';' al comienzo del fichero de Plexil, justo antes de la definición del nodo raíz y en cualquier orden, tal y como puede verse en el ejemplo de la figura 2.20.

Las declaraciones globales son útiles ya que permiten visualizar rápida y cómodamente toda la interfaz externa definida para el plan en un único vistazo, así como comprobar de forma estática la corrección y consistencia de las declaraciones indicadas.

```
1 Command detener();
2 Integer Command   elevar(Integer num, Integer exp);
3 Real Lookup difTemperatura(Real tempAnt);
4 LibraryNode RaizCuadrada(In Real a, In Real b, In Real c,
    InOut Real x1, InOut Real x2);
```

Figura 2.20: Ejemplo de declaraciones globales

2.7. Nodos librería

A las variables declaradas en la interfaz de un nodo se les llama también parámetros. Al definir la interfaz se puede declarar también el tipo de la variable o parámetro. Esto último es útil y obligatorio para lo que se conoce como nodos librería o *library nodes*. Estos nodos no son un tipo de nodo como tal, sino que definen la interfaz necesaria para que los nodos de llamada a librería puedan interactuar con el nodo librería definido que puede ser de cualquiera de los tipos vistos anteriormente.

Los nodos librería deberán especificar tanto la interfaz como el tipo de los datos que serán reemplazados por los valores asignados mediante el nodo de llamada a librería. Además, estos nodos serán el nodo de mayor nivel dentro de la jerarquía del plan en el que estén contenidos[5]. Por tanto, todo nodo superior de un plan podrá ser utilizado como nodo librería y usado para ser llamado desde un nodo librería desde otro plan.

En la figura 2.21 puede verse un ejemplo de un nodo librería y la forma de codificar un nodo de llamada a librería para que interactúe con el primero. Dado que cada fichero PLEXIL sólo contiene un plan y que un plan sólo puede tener un nodo superior, será necesario el empleo de un archivo por cada nodo librería a utilizar más otro para el que implemente los nodos de llamada a librería. Además, habrá que declarar antes de la codificación del plan los nodos librería que vamos

[5]Actualmente sólo se permite un nodo superior o *top level* en un plan.

```
1  --- archivo F.ple ---
2  F:
3  {
4    In Integer i;
5    InOut Integer j;
6    Assignment: j = j * j + i;
7  }
8  --- final F.ple ---
9
10 --- archivo llamadaLibreria.ple ---
11 LibraryNode F(In Integer i, InOut Integer j);
12 llamadaLibreria:
13 {
14   Integer k = 2;
15   LibraryCall: F(i=12, j=k);
16 }
```

Figura 2.21: Ejemplo de nodo librería y nodo de llamada a librería

a usar mediante la correspondiente declaración global. Para ello se hará uso de la notación `LibraryNode <nombre>[(<lista_args>)]`, donde `nombre` es el nombre del nodo raíz del fichero librería (el nombre del fichero se indicará al llamar al ejecutor) y `lista_args` será la lista de parámetros (si es necesaria) que requiere la interfaz del nodo librería usado. Habrá que declararlos indicando su interfaz, tipo (el tipo debe coincidir exactamente, no se permite conversión implícita de enteros a reales, por ejemplo) y nombre de parámetro y separando cada parámetro de la lista por comas. Para llamar al nodo librería bastará con declarar un nodo de llamada a librería en cualquier punto del plan indicando el nombre del parámetro y el valor que se le asignará al realizar la sustitución de valores.

Puede verse que un nodo librería se comporta como una función, pero es extensible al nivel que deseemos, ya que el nodo librería puede ser cualquier tipo de nodo, puede declararse como un nodo lista con una serie de nodos de comando, asignación o la combinación que deseemos, lo que permite incrustar un plan dentro de otro en el punto que deseemos. Esto permite una gran versatilidad a la hora de modelar planes de gran tamaño (permite generar ficheros de menor tamaño y más cómodos de manipular) o ejecutar un plan u otro en base a los estados del plan inicial. A la hora de ejecutar el plan, el nodo librería se expandirá completamente en el plan original y se ejecutará concurrentemente a éste como si se tratase de un nodo más del plan original.

2.8. Instrucciones de alto nivel

En las últimas revisiones de PLEXIL, se han implementado acciones de alto nivel, similares a las utilizadas en otros lenguajes de programación más extendidos. La utilidad de las instrucciones de alto nivel es notable en la simplificación del código, siendo en general una abstracción a la hora de programar los nodos vistos anteriormente.

2.8.1. Secuencia ordenada, *Sequence*

Los nodos descritos dentare de una instrucción de tipo *Sequence* son ejecutados en el mismo orden en el que son descritos. Si alguno de los nodos de la secuencia falla, (estado del nodo *FAILURE*), toda la secuencia terminará mostrando el estado de

FAILURE. La sintaxis de esta acción se puede ver en la 2.22 mostrada a continuación. El núcleo de PLEXIL permite la omisión de la etiqueta `Sequence`, siendo obligatorio el uso de las llaves {...}.

```
Sequence
{
    Nodo1:
            ...
    Nodo2:
            ...
}
```

Figura 2.22: Sintaxis de la instrucción de alto nivel `Sequence`

2.8.2. Secuencia no verificada, *Unchecked Sequence*

La ejecución de esta instrucción es similar a la instrucción `Sequence`, a excepción de que la comprobación de la ejecución en los nodos no es verificada. Ante el fallo de un nodo intermedio, no se detiene la ejecución y se continúa con el siguiente nodo.

```
UncheckedSequence
{
    Nodo1:
            ...
    Nodo2:
            ...
}
```

Figura 2.23: Sintaxis de la instrucción de alto nivel `UncheckedSequence`

2.8.3. Intento (*Try*)

Este tipo de instrucción es derivado de *Sequence* donde se ejecutan los nodos de forma secuencial, hasta que uno de ellos finaliza correctamente. En este caso se abandona la instrucción y el resultado es correcto (`SUCCESS`)

```
Try
{
    Nodo1:
            ...
    Nodo2:
            ...
}
```

Figura 2.24: Sintaxis de la instrucción de alto nivel `Try`

2.8.4. Concurrencia, *Concurrence*

Las instrucciones *Concurrence* son de carácter concurrente o simultáneas. Permiten la ejecución en paralelo de los nodos descritos en su interior. Dado el empleo de PLEXIL en actividades de tiempo real, cabe destacar que el máximo tiempo de

ejecución de los nodos descritos en esta instrucción, será directamente proporcional al nodo más lento. Todas las acciones terminarán a la vez. No obstante, por medio de las *Conditions* vistas anteriormente, se puede modificar el comportamiento de ejecución del nodo, modificando los inicios y finales de los nodos a ejecutar. En el ejemplo de la figura 2.25, `Nodo1` y `Nodo2` se ejecutarán en paralelo, mientras que el `Nodo3` no comenzará su ejecución hasta no finalizar el `Nodo1`.

```
1  Concurrence
2  {
3     Nodo1:
4           ...
5     Nodo2:
6           ...
7     Nodo3:
8           StartCondition Nodo1.state == FINISHED;
9           ...
10 }
```

Figura 2.25: Sintaxis de la instrucción de alto nivel `Concurrence`

2.8.5. Instrucción condicional, *if-then-else*

Este tipo de instrucción evalúa una condición inicial (`if`). Si es correcta (`true`), pasa a ejecutarse los nodos en su interior, en caso de no ser correcta (`false` o `Unknown`), es ignorada.

Se ha implementado a su vez `else` que pasa a su ejecución si la condición anterior no es correcta, o `elseif` que en caso de no ser correcta la condición anterior, pero se cumpla la condición impuesta a continuación. La sintaxis se muestra en la imagen 2.26, donde las condiciones `condition1` y `condition2` son expresiones lógicas de tipo *Bool*.

```
1  if condicion1
2  {
3     ...
4  } elseif condicion2
5  {
6     ...
7  } else
8  {
9     ...
10 }
11 endif
```

Figura 2.26: Sintaxis de la instrucción de alto nivel `If-then-else`

En la figura 2.27 se muestra un ejemplo de ejecución de una instrucción *if-then-else* con dos condiciones evaluadas. Es importante destacar que PLEXIL permite la evaluación de condiciones sin el uso de paréntesis para determinar el resultado. Es de buena práctica emplear estos para una correcta depuración. A modo de ejemplo, se ha planteado el uso anidado de instrucciones de alto nivel, como son *Sequence* y *Concurrence*.

```
1  if dato1>= dato2
2  {
3     Sequence
4     {
5        Nodo1:
6              ...
7        Nodo2:
8              ...
9     }
10 }
11 elseif (dato1 < (dato2 - 5))
12 {
13    Concurrence
14    {
15       Nodo3:
16             ...
17       Nodo4:
18             ...
19    }
20 } else
21 {
22    Nodo5:
23          ...
24 }
25 endif
```

Figura 2.27: Ejemplo de código de la instrucción de alto nivel `If-then-else`

2.8.6. Bucle *While*

Los bucles *While* son instrucciones muy utilizadas para la repetición condicionada de un nodo. Se evalúa la condición inicial, y si es correcto (`true`) se pasa a su ejecución. Al finalizar este, se vuelve a evaluar, y en caso de cumplirse la condición, se repite de forma iterativa. La condición de escape sólo se evalúa al finalizar la iteración anterior, pero puede controlarse con las *EndConditions* vistas en 2.5.

```
1  while condicion
2  {
3     ...
4  }
```

Figura 2.28: Sintaxis de la instrucción de alto nivel `While`

2.8.7. Bucle *for*

Este tipo de bucles evalúan una condición mediante una variable, cuyo valor es modificado en cada iteración. La variable evaluada debe ser de tipo `Integer` o `Real`, y la condición debe dar un resultado de tipo `Bool` (`true`, `false` o `Unknown`). En la figura 2.29 se puede ver un ejemplo de la sintaxis de la instrucción *for*.

En este caso, la documentación de PLEXIL exige el uso de paréntesis para la definición de la instrucción `for`. Cada uno de los campos o parámetros del bucle, se deberán separar por ';'.

```
1  for (TIPO variable = valor_inicial;  variable_condicion;
       modificación variable)
2  {
3      ...
4  }
5
6  // Ejemplo
7  for (Integer i = 0; i <= num_max; i + 1) { ... }
8  for (Real k = 300; k ! = 200; K - 2 * (301-K) ) { ... }
```

Figura 2.29: Sintaxis de la instrucción de alto nivel `For`

2.8.8. Instrucción *OnCommand*

La instrucción *OnCommand* es una llamada a un comando externo al plan de PLEXIL. Se utiliza en configuraciones con múltiples ejecutores donde un ejecutor recibe un comando enviado por otro ejecutor. El principal uso de esta instrucción es la ejecución de comandos remotos entre sistemas. La sintaxis de la instrucción (figura 2.30) requiere del nombre del comando a llamar (`command-name`), una lista de parámetros requeridos para su ejecución (`parámetros`) y un comando de retorno obligatorio (`SendReturnValues(valor)` donde `valor` será `true` en caso de ser omitido el comando de retorno).

```
1  OnCommand <command-name> [ parametros]
2     <action>;
3
4  <command-name>:        // Nodo remoto
5     ...
6     SendReturnValues(<valor>);
```

Figura 2.30: Sintaxis de la instrucción de alto nivel `OnCommand`

2.8.9. Instrucción *OnMessage*

Instrucción similar a `OnCommand`, salvo que se prescinde de parámetros y sólo se recibe el texto devuelto por el mensaje `SendMessage(<string>)`.

```
1  OnMessage <command-name>
2     <action>;
3
4  <command-name>:        // Nodo remoto
5     ...
6     SendMessage(<string>);
```

Figura 2.31: Sintaxis de la instrucción de alto nivel `OnMessage`

2.8.10. Comandos síncronos, *Synchronous Commands*

PLEXIL provee un estilo de comandos de ejecución síncrona donde se toma como referencia el tiempo (`time`). El tiempo (`time`) no es un concepto especial de PLEXIL, es una referencia externa a un estado del mundo. Puede verse en profundidad el uso

del tiempo en el apartado 2.10. En los comandos síncronos, se puede pasar por parámetros un *timeout* y una tolerancia, que indica el tiempo máximo que puede esperar el comando para recibir el valor ejecutado. Superado este tiempo, se finaliza la ejecución. En la figura 2.32 se puede observar varios ejemplos de funcionamiento de la instrucción `SynchronousCommand`.

```
1  \\ Llamada a la funcion sin parámetros
2  SynchronousCommand funcion ();
3
4  \\ Llamada a función con argumentos y un timeout de 5.0
5  \\ unidades de tiempo y tolerancia 0.1 unidades de tiempo
6  SynchronousCommand valorRetornado = funcion(parametro1,
7                        parametro2, ...) Timeout 3.0, 0.1;
```

Figura 2.32: Sintaxis de la instrucción de alto nivel `Synchronous Command`

2.8.11. Instrucción de espera, *Wait*

Instrucción que bloquea la ejecución a la espera de un determinado intervalo de tiempo. Del mismo modo que los comando síncronos, el concepto de `time` en PLEXIL es un estado del mundo, que debe ser cambiado de forma externa al plan desarrollado. En la figura 2.33 se puede ver un ejemplo de aplicación.

```
1  \\ 3.0 unidades de tiempo
2  Real tiempo = 3.0;
3
4  \\ Tolerancia de 0.1 unidades de tiempo
5  Real tolerancia = 0.1;
6
7  Wait tiempo, tolerancia;
```

Figura 2.33: Sintaxis de la instrucción de alto nivel `Wait`

2.9. Posibles estados de los nodos

En PLEXIL los nodos pueden acceder a una serie de estados internos que indican como se encuentra el nodo. Cada nodo podrá acceder a su estado interno, el de sus hermanos, hijos o padres, pero no más lejos. El estado interno de un nodo consiste en varios valores:

- **Estado de ejecución**: determina en que fase de la ejecución se halla el nodo. Para ello se utiliza la clausula `<NodeId>.state` que retornará uno de los siete valores mostrados en la figura 2.34. Los estados posibles son:
  - INACTIVE: la ejecución del plan todavía no ha llegado al nodo, y, por tanto, éste no puede entrar en ejecución y esta inactivo.
  - WAITING: la ejecución del plan ha llegado al nivel del nodo y éste puede entrar en ejecución, pero está a la espera de que se cumpla la condición de inicio para comenzar su ejecución.
  - EXECUTING: una vez el nodo verifica la condición de inicio, pasa a ejecución. Ahora comprobará la precondición y ejecutará el resto del cuerpo del nodo. Este estado se mantiene hasta la finalización del nodo.

  - FINISHING: completada la ejecución del nodo, este pasa a finalizando. Aquí comprueba las condiciones de finalización y postcondiciónes.
  - ITERATION_ENDED: tras comprobar las condiciones de finalización, el nodo pasa a este estado que indica que ha finalizado una ejecución. En caso de tener una condición de repetición, pasará al estado de espera para poder volver a ejecutarse.
  - FINISHED: el estado finalizado indica que el nodo ha terminado su ejecución y que no es apto para volver a ser ejecutado (no tiene condición de repetición).
  - FAILING: este estado indica que se ha producido un error durante la ejecución del nodo y que se abortará la ejecución del mismo.

- **Tiempo de inicio o finalización**: permite conocer el instante de tiempo en el cual ha empezado o finalizado cada uno de los siete posibles estados del nodo. Si el estado o punto de tiempo solicitado no ha ocurrido, entonces devolverá UNKNOWN. Para conocer el tiempo se utiliza una clausula como la que sigue: `<NodeId>.<node state>.<timepoint>` donde `nodestate` corresponde a uno de los siete descritos anteriormente y `tiempoint` será START o END.

- **Estado o valor de salida**: el estado de salida será un valor devuelto por el nodo al finalizar su ejecución. Para conocer este valor se utiliza la clausula `<NodeId>.outcome` y puede devolver uno de los siguientes cuatro valores:

  - SUCCESS: el nodo ha terminado correctamente.
  - FAILURE: el nodo ha fallado en su ejecución.
  - SKIPPED: no se ha ejecutado el nodo ya que se cumplía la condición de omisión.
  - UNKNOWN: el nodo está en ejecución o no se ha ejecutado todavía.

- **Estado de fallo**: en caso de que un nodo haya fallado en se ejecución, mediante la clausula `<NodeId>.failure` se puede conocer la causa de dicho fallo. Ésta nos devolverá uno de los siguientes cinco valores:

  - PRE_CONDITION_FAILED: el nodo no ha podido ejecutarse por no verificar la precondición tras poder entrar en ejecución al verificar la condición de inicio.
  - POST_CONDITION_FAILED: el nodo ha sido ejecutado y ha verificado la condición de finalización pero no la postcondición, y, por tanto, su ejecución es incorrecta.
  - INVARIANT_CONDITION_FAILED: indica que el nodo ha abortado su ejecución por no verificar la condición invariante en algún momento de su ejecución.
  - PARENT_FAILED: indica un fallo en el nodo debido a un fallo en la ejecución de su nodo padre.
  - UNKNOWN: devolverá este valor en caso de haber acabado correctamente.

- **Estado del manejador de comando** (sólo para nodos de comando): este estado permite conocer la última respuesta del manejador de comando asignado al nodo solicitado. Los valores y significados posibles para el manejador de comando se verán en la sección 2.11.2.

En la figura 2.34 pueden verse todos los posibles estados y la sintaxis de los mismos.

```
1  <nodeId>.state <= {INACTIVE, WAITING, EXECUTING, FINISHED,
       ITERATION_ENDED, FAILING, FINISHING}
2
3  <nodeId>.<nodestate>.<{START, END}> <= <marca de tiempo>
4
5  <nodeId>.outcome <= {SUCCESS, FAILURE, SKIPPED, UNKNOWN}
6
7  <nodeId>.failure <= {INVARIANT_CONDITION_FAILED,
       POST_CONDITION_FAILED, PRE_CONDITION_FAILED,
       PARENT_FAILED, UNKNOWN}
8
9  <nodeId>.command_handle <= {COMMAND_ACCEPTED, COMMAND_SUCCESS
       , COMMAND_RCVD_BY_SYSTEM, COMMAND_SENT_TO_SYSTEM,
       COMMAND_FAILED, COMMAND_DENIED, UNKNOWN}
```

Figura 2.34: Posibles estados de los nodos

2.10. Mundo exterior

PLEXIL puede comprobar el estado del mundo exterior mediante operaciones de búsqueda o *lookups*. Esta operación complementa a las ya vistas que permiten interactuar modificando el mundo exterior (comandos). Los *lookups* permiten comprobar el estado del mundo de una forma rápida y sencilla, con una particularidad: sólo pueden utilizarse en las condiciones que rigen el control de ejecución de los nodos. Esto será muy útil para poder indicar que ciertos nodos se ejecuten ante un evento particular del mundo exterior o, por el contrario, detengan su ejecución.

La lectura de los estados del mundo exterior se realizará mediante el uso de un *lookup* sobre el nombre de un dominio especifico para el cual tendremos un control de bajo nivel que devuelva el dato asociado al estado. Por tanto, de forma general, un *lookup* estará asociado al control de un sensor del sistema a controlar. Habrá que distinguir entre dos tipos de *lookups*:

- **Comprobación inmediata**: permite conocer el valor del estado de forma inmediata. Para ello, se usa una clausula del tipo `LookupNow (<nombre_estado>)`. Ésta sólo podrá aparecer en las *check conditions* (precondición, postcondición o condición invariable) o en el cuerpo de los nodos de comando. Es muy útil para comprobar antes, durante y después que una acción se lleve a cabo en ciertas condiciones de seguridad.

- **Comprobación por cambio**: esta comprobación esta basada en eventos y retorna el valor inicial del estado, y, tras ello, los siguientes valores según se vayan produciendo cambios en el estado. Se emplea la clausula `LookupOnChange (<nombre_estado>, [<tolerancia>])` para indicar el estado sobre el cual se quiere realizar un seguimiento. En dicha clausula se deberá indicar el nombre del estado, y, opcionalmente, la tolerancia (debe ser un valor entero o real, si no se especifica toma 0 por defecto). La tolerancia indicará el valor mínimo de cambio que se debe dar en el estado para que se devuelva el nuevo estado. Este tipo de consulta sólo puede aparecer en las *gate conditions* (condición de inicio, final, repetición u omisión). Por tanto, permite controlar cuando un nodo puede ejecutarse o finalizar la ejecución en base al estado del mundo.

A todos los efectos un *lookup* es una función (con al menos un parámetro de tipo cadena de caracteres para el nombre del estado) que devolverá un valor de un tipo simple que podremos almacenar (sólo en el caso de `LookupNow` en un nodo

de comando), o utilizar en expresiones booleanas para controlar las condiciones de ejecución del nodo. Su uso es necesario ya que, al contrario que las llamadas a funciones que sólo pueden aparecer en el cuerpo de un nodo de llamada a función, su uso fundamental está en las condiciones de los nodos. En la figura 2.35 puede verse un ejemplo del uso de *lookups*.

```
1 bajarTemperatura:
2 {
3     StartCondition: LookupOnChange("Temperatura") > 90;
4     PreCondtion: LookupNow("VentiladorOperativo");
5     EndCondition: LookupOnChange("Temperatura",5) < 60;
6     PostCondition: LookupNow("Temperatura") < 90;
7     RepeatCondition: true;
8     Command: activarVentilador(1);
9 }
```

Figura 2.35: Ejemplo del uso de *lookups*

2.10.1. Control del tiempo

PLEXIL no dispone de un concepto nativo del tiempo, sin embargo, se puede trabajar con el como un estado del mundo. Dicho estado está predefinido con el nombre de `time` y permite operar con él ya que se trata de un valor real.

2.11. Gestión de recursos

PLEXIL soporta la utilización y gestión de recursos dentro del plan. Actualmente, esto es sólo aplicable a los nodos de comando, ya que serán estos los que realicen cambios en el sistema, y, por tanto, serán potencialmente consumidores de recursos. El uso de recursos será comprobado en tiempo de ejecución por medio de una entidad desarrollada a tal fin: el arbitrador de recursos.

Los recursos son entidades limitadas que puntualmente pueden ser recurridas para realizar ciertas acciones que requieran de su uso o consumo. PLEXIL soportará recursos unarios, no unarios, jerárquicos y recursos renovables. En todos los casos, los recursos pueden ser o no consumibles, y reutilizables en función de las necesidades.

Como ya se ha indicado, los recursos sólo podrán ser especificados para su uso en los nodos de comando. Para crear un recurso, sólo es necesario indicar el uso que se le va a dar en un nodo de comando y el sistema se encargará de gestionar la instancia del recurso y todas las apariciones del mismo para controlar su uso. Además, dentro de un mismo nodo, se podrán utilizar tantos recursos como sea necesario para ejecutar el comando. En la figura 2.36 pueden verse los elementos necesarios para el uso de recursos, así como un ejemplo del uso de los mismos. A continuación se detalla cada elemento del uso de recursos:

- `ResourcePriority`: indica la prioridad en el uso del recurso que va inmediatamente después. Sólo puede haber una indicación de prioridad en cada nodo, independientemente del número de recursos implicados. La prioridad vendrá indicada por un valor entero sin signo, siendo más prioritario el número de menor valor.
- `Resource`: especifica las características del recurso a emplear. Consta de los siguientes elementos:

- `Name`: indica el nombre (mediante una cadena de texto) que identifica al recurso necesario. Es el único elemento obligatorio al declarar un recurso.
- `LowerBound`: para recursos no unarios, permite definir el valor mínimo que será requerido del recurso para la realización del comando. Será representado por un valor real.
- `UpperBound`: define el máximo de un recurso que podrá consumir el nodo para ejecutar el comando. Está representado por un valor real. En caso de que deseemos consumir una cantidad fija y no definir un rango, tanto `LowerBound` como `UpperBound` deberán tener el mismo valor. Si, además, ese valor es de signo negativo en ambos, significará que es un recurso renovable y se generará la cantidad indicada en valor absoluto de dicho recurso.
- `ReleaseAtTermination`: este valor booleano indica si el recurso volverá a estar disponible tras la ejecución del nodo en el cual se hace uso de él. Por defecto, el recurso es desbloqueado permitiendo así su uso por otros nodos, si esta propiedad se especifica a falso, se tratará de un recurso consumible que una vez usado no pueda ser utilizado por otros nodos.

```
1  ResourcePriority:
2    (obligatorio: tipo = int, mayor prioridad al menor valor);
3  Resource:
4    Name = (obligatorio: tipo = string),
5    LowerBound = (opcional: tipo=real, defecto=0.0),
6    UpperBound = (opcional: tipo=real, defecto=0.0),
7    ReleaseAtTermination = (opcional: tipo=bool, defecto=true);
8
9  ejemploRecursos:
10 {
11    Real maximo = 20.0;
12    ResourcePriority: 10;
13    Resource: Name = "brazo_der";
14    Resource: Name = "brazo_izq";
15    Resource: Name = "memoria",
16     LowerBound = 10.0,
17     UpperBound = maximo,
18     ReleaseAtTermination = false;
19    Command: comando();
20 }
```

Figura 2.36: Utilización de recursos y ejemplo

Al declarar los recursos necesarios por un nodo, se declarará la prioridad del nodo para el uso de recursos, y, después, una lista de los recursos a usar mediante la declaración sucesiva de etiquetas `Resource` con los elementos internos necesarios que deseemos. La lista de elementos internos del recurso irá separada por comas, salvo el último elemento, que acabará con ';' como indicador de final de instrucción.

Los valores para los elementos de los recursos son parametrizables. Por tanto, podremos declarar variables de cadenas de caracteres para reemplazar el literal que especifica el nombre, utilizar valores reales como límites en el uso de recursos (lo que permite un uso dinámico en tiempo de ejecución), o una variable booleana para indicar si el recurso es renovable o no.

2.11.1. Arbitrador de recursos

Todos los comandos identificados en el ciclo de ejecución en curso (aquellos nodos que pueden ser ejecutados en ese instante), serán enviados al arbitrador de recursos en vez de ir directamente al subsistema encargado de su ejecución. El arbitrador será el encargado de analizar el uso de los recursos que hace el grupo de los nodos de comando recibido, de tal forma que si se dan las condiciones necesarias en el uso de recursos, éste entregará los comandos al subsistema encargado de su ejecución, indicando dicha circunstancia al manejador del comando. En caso de haber un conflicto de recursos, los nodos que no puedan ser ejecutados, recibirán un indicador en el manejador de comandos (véase la siguiente sección) para indicar que la ejecución del comando ha sido denegada.

El arbitrador de recursos se verá más en profundidad en el capítulo 3, ya que éste forma parte del ejecutor *Universal Executive*.

2.11.2. Manejador de comando

Como ya se indicó, cada comando (y por tanto, cada nodo de comando) tiene asociado un manejador de comando. El manejador de comando a efectos prácticos es simplemente un valor de estado accesible por el nodo y sus parientes cercanos. Este estado será establecido inmediatamente por el arbitrador de recursos cuando el comando sea puesto en ejecución. Es importante que al iniciar la ejecución del comando se establezca un valor particular para el manejador asociado, ya que eso es el comienzo de la etapa de ejecución del comando.

Al solicitar la ejecución de un comando pueden suceder varias cosas: que el arbitrador rechace el comando, o que éste sea aceptado y se envié al subsistema de ejecución del comando. Una vez ejecutado, se asignará un nuevo valor al manejador que indique como fue el resultado de la ejecución del comando. Por tanto, en todo momento se podrá conocer el estado del comando y, en base a eso, puede controlarse la ejecución de otros nodos. Cuando un nodo de comando está listo para su ejecución, el manejador del comando asociado podrá recibir uno o más de los siguientes valores:

- COMMAND_ACCEPTED: el arbitrador de recursos (en caso de que el comando requiera recursos) ha comprobado que es posible la ejecución del comando y éste va a ser enviado al subsistema para su ejecución.
- COMMAND_DENIED: el comando no puede ejecutarse debido a que el arbitrador ha comprobado que los recursos necesarios no están disponibles.
- COMMAND_SENT_TO_SYSTEM: el comando ha sido enviado al subsistema externo para su ejecución.
- COMMAND_RCVD_BY_SYSTEM: el sistema externo ha recibido el comando.
- COMMAND_SUCCESS: el comando ha sido ejecutado correctamente.
- COMMAND_FAILED: la ejecución del comando ha sido incorrecta.
- UNKNOWN: en caso de que el comando se esté ejecutando.
- COMMAND_ABORTED: abortado por el ejecutor cuando la condición invariante falle.
- COMMAND_ABORT_FAILED: fallo al abortar la ejecución del nodo tras evaluar negativamente la condición invariante.

Los nodos de comando tienen una condición de finalización predefinida que se comporta de la forma que se muestra en la figura 2.37 en función de si el usuario ha definido o no una condición de finalización particular para el nodo.

```
1 if <existe EndCondition definida por el usuario>
2    COMMAND_DENIED or (COMMAND_ACCEPTED and <EndCondition>)
3 else
4    COMMAND_DENIED or COMMAND_ACCEPTED
```

Figura 2.37: Condición de finalización de un nodo de comando

2.12. Comentarios

Los comentarios en Plexil pueden ser para una línea o para un bloque como se muestra en la figura 2.38.

```
1 // Comentario de una línea
2 /*
3  Comentario de un bloque
4  Texto comentado
5 */
6
7 Comment "Comentario de una linea o bloque, delimitado por
      dobles comillas";
```

Figura 2.38: Comentarios en Plexil

2.13. PlexilScript

PLEXIL está preparado para interactuar con un sistema de bajo nivel que opere sobre el mundo y devuelva resultados a través del ejecutor para que este actualice los datos del plan. Dado que esto implica un sistema complejo y los planes pueden contener errores que deben ser depurados antes de integrarlos a un sistema complejo, PLEXIL dispone de un lenguaje de scripts de simulación compresible por el *Universal Executive*, que, al igual que el lenguaje PLEXIL, pueden escribirse en Plexil (PLEXIL estándar) o PlexilLisp, pero finalmente se codificarán en XML para ser leídos por el ejecutor. La traducción entre la sintaxis que se verá aquí (Plexil estándar) y XML se verá en la siguiente sección.

Los scripts deberán ser capaces de responder como lo haría el subsistema de control de bajo nivel. Por tanto, deberá ser capaz de responder a *lookups* y comandos. Además, si el tiempo es importante, y dado que el tiempo se gestiona como un estado del mundo, los scripts también podrán controlar el paso del tiempo.

Un script de simulación de PLEXIL se compondrá de dos bloques, pudiendo estar vacíos en función de las necesidades de la simulación. Estos bloques serán los siguientes:

- **Estado inicial**: el estado inicial del mundo describe los valores iniciales de los estados. No siempre será necesario iniciar los estados, ya que un estado no inicializado tendrá un valor UNKNOWN.

- **Variaciones del mundo**: aquí se describirá linealmente como progresa el mundo. En este punto se podrán realizar cambios en los valores de los estados y las respuestas dadas por los comandos, así como el estado que debe recibir el manejador de estados de los mismos. Los *lookups* recibirán el último valor escrito para el estado solicitado, por tanto, si se desea gestionar el tiempo, sólo será necesario ir actualizando el valor del estado predefinido `time` con el valor deseado, siempre de forma progresiva.

Un script de Plexil tendrá la forma mostrada en la figura 2.39. Los dos elementos descritos anteriormente se describen mediante las clausulas `initial-state` y `script` para el estado inicial y la variación del mundo, respectivamente.

```
1  initial-state {
2     state Posición("Destino" : string) = false : bool;
3     state time() = 0 : real;
4  }
5  script {
6     state time() = 8 : real;
7     state Posición ("Destino" : string) = true : bool;
8     command-success avanzar (1.0 : real);
9     state time() = 9 : real;
10    command-accepted tomarMuestra ();
11 }
```

Figura 2.39: Ejemplo de script de simulación de Plexil

A continuación se describirá la sintaxis de los elementos que pueden aparecer. Primero se verá la modificación de los estados, los cuales pueden aparecer tanto en el estado inicial como en el script. Después se verán los elementos que sólo pueden formar parte del script que son respuestas a comandos. Hay que destacar que los nombres de los tipos no tienen capitalización (al contrario que en Plexil) y que el nombre de los tipos `Boolean` e `Integer` se sustituyen por `bool` e `int` respectivamente.

- **Modificación de los estados**: la clausula que permite tanto definir un estado inicial, como variar los mismos durante el script de simulación es la mostrada en la figura 2.40. Tras la etiqueta `state` deberá indicarse el nombre del estado a modificar, así como la lista de parámetros que identifican los datos del estado a comprobar. Ésta será una lista de pares valor-tipo separada por comas, pudiendo estar vacía si no se precisan parámetros. Finalmente se indicará el valor de retorno de la consulta de estado y su tipo. Cada vez que se realice una consulta a un estado, se devolverá el último estado actualizado.

```
1  state <estado> ([<parámetro> : <tipo_param>]*) = <resultado>
       : <tipo_res>;
```

Figura 2.40: Modificación de los estados en Plexilscript

- **Comandos**: la definición de los comandos se divide en dos tipos de clausulas. La primera permite responder a un comando con un valor de retorno, y, la segunda ofrece la posibilidad de establecer el valor para el manejador del comando. Ambas clausulas tendrán la misma sintaxis que la modificación de

estados, salvo por la etiqueta empleada y que el segundo grupo no permite generar un valor de retorno al ser sólo un modificador del estado del manejador. Las etiquetas y las clausulas completas para cada grupo de respuesta a comando pueden verse en la figura 2.41. Para el segundo grupo, las etiquetas corresponden a los posibles estados del manejador de comandos vistos en la sección 2.11.2.

```
1  {command, command-abort, command-ack} <comando> ([<parámetro>
       : <tipo>]*) = <valor> : <tipo>;
2
3  {command-accepted, command-denied, command-sent-to-system,
     command-rcvd-by-system, command-success, command-failed}
     <comando> ([<parámetro> : <tipo>]*);
```

Figura 2.41: Retorno a un comando y al manejador asociado en Plexilscript

Para los comandos se deberán especificar correctamente los parámetros con el mismo valor que sus respectivas llamadas en el plan, ya que en caso de no coincidir la lista de parámetros indicada en el script con la lista utilizada al realizar el comando, se provocará un fallo en tiempo de ejecución al enviar una respuesta a un comando que no ha sido lanzado desde el plan. En todo caso (incluyendo la modificación de los estados), los valores de retorno deberán ser un tipo simple de los admitidos por Plexil.

La sintaxis completa para la codificación de scripts en Plexil se puede ver en la figura 2.42

```
1  elemento =
2      script         { <elemento> ... }
3    | initial-state  { <elemento> ... }
4    | simultaneous   { <elemento> ... }
5    | update-ack     <nombre> ;
6    | state          <nombre> (<valor> : <tipo>, ...) = <valor>
       : <tipo> ;
7    | command        <nombre> (<valor> : <tipo>, ...) = <valor>
       : <tipo> ;
8    | command-abort  <nombre> (<valor> : <tipo>, ...) = <valor>
       : <tipo> ;
9    | command-ack    <nombre> (<valor> : <tipo>, ...) = <valor>
       : <tipo> ;
10   | command-accepted        <nombre> (<valor> : <tipo>, ...);
11   | command-denied          <nombre> (<valor> : <tipo>, ...);
12   | command-sent-to-system  <nombre> (<valor> : <tipo>, ...);
13   | command-rcvd-by-system   <nombre> (<valor> : <tipo>, ...);
14   | command-success         <nombre> (<valor> : <tipo>, ...);
15   | command-failed          <nombre> (<valor> : <tipo>, ...);
16 tipo = bool | int | real | string
17      | bool-array | int-array | real-array | string-array
```

Figura 2.42: Sintaxis completa para los scripts de Plexil

2.14. Ejecución de los programas de PLEXIL

Los programas escritos en Plexil (PLEXIL estándar), tendrán por defecto la extensión `.ple`. Dichos ficheros tendrán la sintaxis desarrollada a lo largo de este

capítulo, sin embargo, el formato "comprensible" de PLEXIL para su ejecución es un fichero XML. Dicho fichero XML, cuya extensión será `.plx`, es el que utilizará el UE para ejecutar el plan. Para la obtención de dicho fichero se recurre a un programa que comprueba la corrección del fichero original y genera un nuevo fichero (con el nombre original sustituyendo la extensión) de PLEXIL en XML con el plan traducido a dicho formato. Para la traducción a XML de los planes de Plexil, se recurre a un `shell-script` que invocará al componente encargado de la traducción. La orden que se deberá ejecutar es la mostrada en la figura 2.43. En la figura también se muestra el resultado de la traducción. En caso de que el archivo contenga errores, se mostrará una lista indicando la relación de los mismos.

```
1 $ plexilc archivo.ple
2 PlexilParser version 0.4
3
4 Translating:
5   archivo.ple
6 Writing Core PLEXIL to archivo.plx
7 Done.
8 $
```

Figura 2.43: Traducción de Plexil a XML

El script permite varias opciones que pueden verse escribiendo `plexilc` sin argumentos. La más útil de ellas es la opción `-o` seguida de un nombre de archivo. Dicha opción permite escribir el archivo XML en el archivo indicado en sustitución del comportamiento genérico.

Al igual que los planes, los scripts de simulación deberán ser traducidos a XML. El procedimiento es el mismo que para los archivos de planes. La orden a ejecutar y la salida generada se muestran en la figura 2.44. En este caso, sólo mostrará una salida si hay errores, no dando ninguna información en caso de que la traducción sea correcta.

```
1 $ plexilc archivo-script.pst
```

Figura 2.44: Traducción de los scripts de Plexil a XML

Los archivos de Plexilscript tendrán extensión `.pst`[6] y, tras su traducción a XML, ésta pasará a `.plx`. Para el nombrado de los archivos de scripts se recurre a una convención mediante la cual el nombre del archivo será el nombre del fichero del plan (sin extensión), seguido por `-script` o `_script` y la extensión del archivo en base a su formato. Dicha convención permite que el propio ejecutor localice el script correspondiente al plan a ejecutar siempre que se encuentren en la misma carpeta, evitando así tener que indicar dicho fichero como parámetro.

Una vez se disponga de un plan PLEXIL en formato XML y su respectivo script de simulación (si es necesario), se podrá realizar la ejecución mediante el UE. La descripción del UE viene dada a lo largo del capítulo 3. Si se desea conocer únicamente los detalles necesarios para la ejecución de simulaciones, consulte la sección 3.2 de dicho capítulo.

[6] Anteriormente compartian la extensión `.ple`, pero en las últimas versiones es necesario reemplazar dicha extensión para poder realizar correctamente la traducción a XML.

Capítulo 3

El *Universal Executive*

En este capítulo veremos el sistema encargado de ejecutar los planes escritos en PLEXIL, el UE. Para ello se verá el framework que implementa los elementos necesarios para desarrollar nuestro propio sistema de control y se dará una descripción más completa del arbitrador de recursos, ya que es un elemento fundamental del UE. Además, se verá la aplicación TestExec, que permitirá realizar simulaciones de planes mediante el uso de scripts de PlexilScript.

3.1. *Universal Executive* y TestExec

El *Universal Executive* (UE) es un sistema que interpreta y ejecuta los planes escritos en PLEXIL. Técnicamente el UE es una librería que implementa PLEXIL. Por tanto, para implementar el sistema de control mediante el UE, tendremos que disponer de un soporte de bajo nivel adecuado a nuestro hardware para ejecutar los comandos necesarios, el cual se comunicará a través de una interfaz definida con el framework del UE. En esencia, el UE es un framework que nos permitirá cargar y ejecutar planes de PLEXIL dentro de nuestro propio programa de control y que éste se comunique con el soporte de bajo nivel a través de una interfaz definida en dicho framework. En la figura 3.1 pueden verse gráficamente los elementos básicos necesarios para diseñar un sistema de control mediante PLEXIL y el UE. El funcionamiento del framework se verá en la sección 3.5.

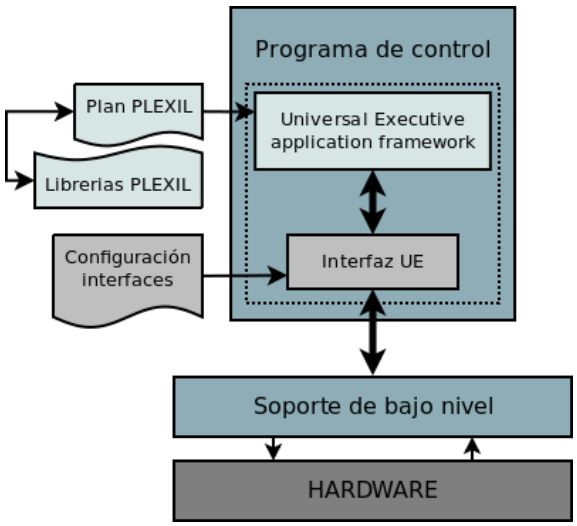

Figura 3.1: Esquema de un sistema controlado por PLEXIL y el UE

A parte de dicho framework, junto a la distribución de PLEXIL se distribuye una aplicación denominada TestExec que implementa el framework del UE. Dicha aplicación permite probar los planes escritos en PLEXIL (con formato XML) y simular el comportamiento del mundo exterior a través de un script PlexilScript. La función de esta aplicación es permitir comprobar el funcionamiento del plan desarrollado para cualquier situación, dado que en el script podemos indicar todas las posibles contingencias que pudieran darse durante la ejecución en el sistema real. Además, permitirá realizar las pruebas sin necesidad de tener que diseñar ni compilar el framework, por lo que es muy útil para el aprendizaje del lenguaje PLEXIL. En la figura 3.2 puede verse un esquema de la aplicación TestExec.

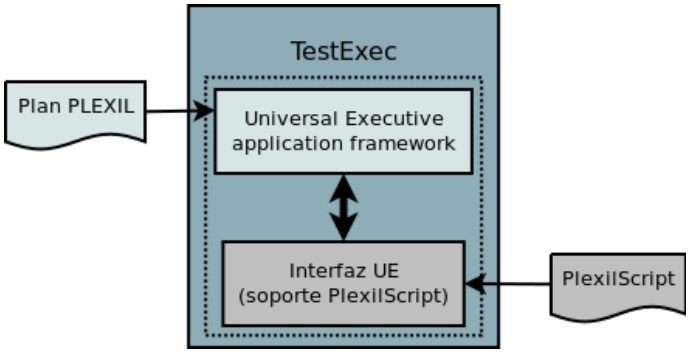

Figura 3.2: Esquema de la aplicación TestExec

3.2. Ejecución de TestExec

Para llevar a cabo la ejecución de TestExec se utiliza la sintaxis mostrada en la figura 3.3.

```
1 run-ue [-s] [-d archivo-debug] plan.plx [plan-script.plx]
2    [-l libreria.plx]*
```

Figura 3.3: Ejecución de TestExec vía script

La opción `-s` omite la información de los archivos usados mostrada al inicio de la ejecución. Con la opción `-d` se puede especificar el archivo de depuración que se deberá utilizar. Dicho archivo permite indicar al ejecutor que información debe mostrar a través del terminal, como la transición entre estados de los nodos, estados de salida, información de los manejadores de comandos, etc. Si no se especifica ningún archivo, por defecto utilizará `Debug.cfg` y, en caso de no existir, no mostrará información durante la ejecución. El contenido de estos archivos se verá en la sección 3.2.1.

Tras las primeras opciones, se deberá indicar que plan deseamos ejecutar. Se deberá indicar el archivo con el plan PLEXIL en formato XML (cuya extensión por defecto es `.plx`). Inmediatamente después del plan se deberá especificar el archivo de script (también en XML) para realizar la simulación. En caso de no indicarlo, buscará en el directorio de trabajo un archivo de script cuyo nombre sea el nombre del plan seguido por `-script` o `_script` para usarlo, si no encontrará ninguno, utilizaría un script genérico de nombre `empty-script.plx`, el cual es un script vacío. En caso de que nuestro plan contenga llamadas a funciones, comandos o consulta de

estados, no se podrá utilizar el script vacío y la ejecución de dicho plan sin su script correspondiente generará un error de ejecución. Finalmente se deberá indicar la lista de librerías que se utilizarán en nuestro plan (sólo si se usan nodos de llamada a librería). Está lista será una sucesión formada por el par `-l` seguido del nombre de archivo librería en XML (con extensión `.plx`).

Este método de ejecución permite ejecutar TestExec a través de un `shell-script`. En la figura 3.4 se muestra el comando necesario para ejecutar TestExec. La sintaxis será la misma que para el caso anterior.

```
1 test-exec_g_rt [-d archivo-debug] -p plan.plx
2    -s plan-script.plx [-l libreria.plx]*
```

Figura 3.4: Ejecución de TestExec

En la figura 3.5 se muestra un ejemplo de ejecución (no se muestra la salida completa). El archivo de depuración utilizado está en la figura 3.6.

3.2.1. Archivos de depuración (*debug*)

Como se ha indicado anteriormente, para que el ejecutor muestre información de interés por el terminal (ya sea simulando el plan con TestExec o en un sistema real), se requiere un archivo de depuración. En el caso de TestExec, dicho archivo deberá estar en la carpeta donde estemos ejecutando la aplicación, y, por defecto, tendrá como nombre `Debug.cfg`, pudiendo indicar otro archivo mediante la opción `-d`.

Un archivo de depuración es un fichero de texto compuesto por líneas. Una línea puede ser un comentario si comienza con el carácter '#'. Para definir los datos que queremos que muestre el ejecutor, deberemos especificar una línea que comience por ':' seguida de una etiqueta. Esta etiqueta consistirá en una cadena de caracteres sin espacios que indicará el tipo de información a mostrar. Generalmente las etiquetas consisten en un par de identificadores separados por ':'. El primero indicará el origen de la información (nodos, procesos del ejecutor, información del manejador de comandos, etc), y el segundo la información en particular. Con ello podremos obtener información del estado de salida de los nodos, la inclusión de un nodo librería en el plan en ejecución, los comandos aceptados para ejecución y mucho más. El archivo de depuración básico es el mostrado en la figura 3.6, el cual mostraría las transiciones de los nodos y sus estados de salida.

Para ver todas las etiquetas disponibles para los archivos de depuración, consulte el archivo:
`$PLEXIL_HOME/universal-exec/Utils/test/CompleteDebugFlags.cfg`

3.2.2. Luv: visor gráfico para TestExec

Luv o *Lightweight Universal Executive Viewer*, es un entorno gráfico para visualizar planes escritos en PLEXIL y realizar ejecuciones de simulación mediante TestExec. Está escrito en Java y su funcionamiento es simple y muy intuitivo. Para ejecutar Luv puede realizarse mediante la interfaz gráfica del sistema operativo[1] o mediante una `shell` tal y como se indica en la figura 3.7. En la figura, la segunda instrucción sólo será posible ejecutarla si se han definido correctamente las variables de entorno de PLEXIL.

[1] Si al intentar ejecutar el plan se genera un fallo en Luv, habrá que definir las rutas a las variables de PLEXIL en el entorno de trabajo del sistema gráfico

```
/home/user/test$ run-ue -p lista.plx -l lib.plx -l raiz.plx
Running UE from /home/user/plexil
  Plan:       lista.plx
  Script:     /home/user/plexil/apps/TestExec/scripts/empty-script.plx
  Libraries: -l lib.plx -l raiz.plx

[Node:transition]Transitioning 'root' from INACTIVE to WAITING
[Node:transition]Transitioning 'root' from WAITING to EXECUTING
[Node:transition]Transitioning 'LeerX' from INACTIVE to WAITING
[Node:transition]Transitioning 'Accion' from INACTIVE to WAITING
[Node:transition]Transitioning 'comprobarEjecucionLib' from INACTIVE to
    WAITING
[Node:transition]Transitioning 'InformarCorrecto' from INACTIVE to
    WAITING
[Node:transition]Transitioning 'InformarFallo' from INACTIVE to WAITING
[Node:transition]Transitioning 'comprobarEjecucionLib' from WAITING to
    EXECUTING
[Node:transition]Transitioning 'LeerX' from WAITING to EXECUTING
[Node:transition]Transitioning 'multiplicar' from INACTIVE to WAITING
[Node:transition]Transitioning 'calcularRaiz' from INACTIVE to WAITING
[Node:transition]Transitioning 'LeerX' from EXECUTING to ITERATION_ENDED
[Node:transition]Transitioning 'LeerX' from ITERATION_ENDED to FINISHED
[Node:outcome]Outcome of 'LeerX' is SUCCESS
   [...]
[Node:transition]Transitioning 'lib' from EXECUTING to FINISHING
[Node:transition]Transitioning 'lib' from FINISHING to ITERATION_ENDED
[Node:transition]Transitioning 'lib' from ITERATION_ENDED to FINISHED
[Node:outcome]Outcome of 'lib' is SUCCESS
[Node:transition]Transitioning 'Accion' from FINISHING to
    ITERATION_ENDED
[Node:transition]Transitioning 'InformarCorrecto' from WAITING to
    EXECUTING
[Node:transition]Transitioning 'InformarFallo' from WAITING to FINISHED
[Node:outcome]Outcome of 'InformarFallo' is SKIPPED
[Node:transition]Transitioning 'Accion' from ITERATION_ENDED to FINISHED
[Node:outcome]Outcome of 'Accion' is SUCCESS
[Node:transition]Transitioning 'InformarCorrecto' from EXECUTING to
    ITERATION_ENDED
[Node:transition]Transitioning 'InformarCorrecto' from ITERATION_ENDED
    to FINISHED
[Node:outcome]Outcome of 'InformarCorrecto' is SUCCESS
[Node:transition]Transitioning 'root' from EXECUTING to FINISHING
[Node:transition]Transitioning 'root' from FINISHING to ITERATION_ENDED
[Node:transition]Transitioning 'root' from ITERATION_ENDED to FINISHED
[Node:outcome]Outcome of 'root' is SUCCESS
/home/user/test$
```

Figura 3.5: Ejemplo de la salida generada por TestExec

```
# Archivo Debug.cfg
:Node:transition
:Node:outcome
```

Figura 3.6: Archivo de depuración utilizado en la figura 3.5

```
$ java -jar $PLEXIL_HOME/src/luv/luv.jar &
$ plexil
$
```

Figura 3.7: Ejecución de Luv

Este capítulo no representa un manual de usuario de Luv, sino una descripción breve de las capacidades que esta aplicación ofrece. Para consultar un manual de

uso detallado, consulte el capítulo 3 (*Viewing Plan Execution with Luv*) en [3].

Luv funciona en modo cliente-servidor comunicando la interfaz gráfica de usuario con TestExec que será el encargado de ejecutar los planes y entregar los resultados a Luv para que este los muestre. Es decir, Luv es una interfaz gráfica para TestExec. Esto implica que podrá trabajar tanto en la máquina local, como por terminal remota. Para ello, tanto Luv como TestExec tienen el debido soporte para comunicación remota, y, por defecto, esta comunicación se llevará a cabo a través del puerto 9787. A continuación, se detallan los dos modos de ejecución de Luv:

- **Máquina local:** sólo es necesario ejecutar Luv como se indica en la figura 3.7. Tras ello, se deben seguir lo siguientes pasos:
  1. Abrir el plan deseado desde el menú "File/Open Plan".
  2. Abrir el script de simulación desde el menú "File/Open Script".Esto no será necesario si el nombre del script se adecua a la convención vista en la sección 3.3.
  3. En caso de no especificar un script o que no se encuentre éste, Luv usará un script vacío creado automáticamente.
  4. Si se desea ejecutar el plan por pasos, se deberá activar la opción "Run/Enable Breaks ".
  5. Proceder a ejecutar el plan: "Run/Execute Plan".
- **Ejecución vía terminal remoto:** mediante este sistema, se ejecutará TestExec por medio del script run-ue y se comunicará con Luv por el puerto antes indicado. Para ello hay que seguir los siguientes pasos:
  1. Ejecutar Luv tal y como se indica en la figura 3.7.
  2. Ejecutar el script run-ue de la forma mostrada en la figura 3.3, anteponiendo la opción `-v` al nombre del plan. Además, podrán ser necesarias las siguientes opciones que se espcificarán antes del plan:
     - Si se desea ejecutar por pasos se deberá utilizar la opción `-b`.
     - Para ejecutar Luv y el UE desde diferentes máquinas, habrá que especificar el nombre de la máquina tras la opción `-h`.
     - Si el puerto de Luv ha sido modificado, se deberá de indicar el nuevo puerto a utilizar tras la opción `-p`.
  3. Una vez ejecutado con las opciones convenientes, en Luv aparecerá el plan deseado y a través de la interfaz gráfica podremos ir ejecutando paso a paso (si hemos activado la opción conveniente) y visualizando los datos relacionados con el plan.

En la figura 3.8 podemos ver el aspecto que tiene la ventana principal de Luv. Junto a ella, se muestra la ventana de debug. Dicha ventana mostrará el resultado que obtendríamos si estuviésemos ejecutando el plan en una terminal. En caso de la ejecución remota, en el terminal remoto se mostraría la misma información que en dicha ventana.

A continuación se describen las opciones que permite Luv:

- Permite pausar la ejecución del plan y reanudarla. Además, permite la ejecución paso a paso del plan o detener la ejecución en los nodos deseados mediante puntos de ruptura. Los puntos de ruptura podrán definirse en base al cambio de estado del nodo, al encontrarse el nodo en un estado en particular, por un estado de salida o por un estado de fallo.

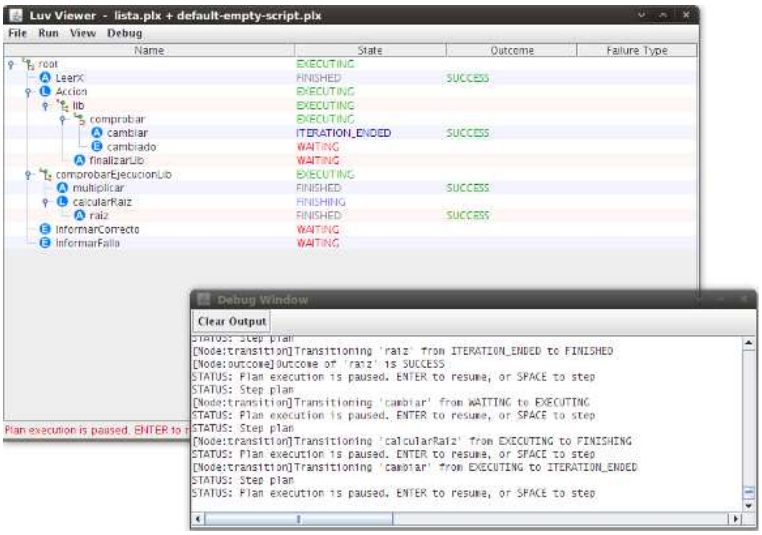

Figura 3.8: Ventana principal y de debug de Luv

- Muestra los nodos en la estructura de árbol del plan, pudiendo mostrar u ocultar las ramas según nuestras necesidades. También permite mostrar u ocultar nodos por nombre o por tipo de nodo y la búsqueda de nodos por nombre.
- Puede generar archivos de depuración con los elementos deseados a través de una interfaz gráfica.
- Permite obtener toda la información relativa a un nodo. La información proporcionada vendrá dada en función del tipo de nodo y de los datos de que disponga, informando así de las variables declaradas y su valor (sólo el valor inicial), las condiciones definidas, y los datos asociados al tipo de nodo (comando, asignación, etc).

3.3. Funcionamiento del *Universal Executive*

En esta sección vamos a describir el funcionamiento del UE para ejecutar los planes escritos en PLEXIL. Para ello se dará una descripción de como transcurre el estado de todos los nodos (y, por tanto, del plan en sí) en el tiempo sin entrar en detalle de los algoritmos que llevan a cabo tareas de más bajo nivel, como por ejemplo la sincronización ante condiciones de carrera. Además, una parte importante (aunque como ya se dijo, externa al UE) para la ejecución es el arbitrador de recursos. El funcionamiento de éste se verá en la sección 3.4.

La ejecución del UE se lleva a cabo a través de ciclos de ejecución que especifican qué, cómo y cuándo variará el estado de los nodos en un instante específico del tiempo. Cabe destacar que la ejecución del plan procederá en pasos discretos y que los eventos (cambios en el mundo que afecten a condiciones, valores de retorno de funciones o comandos, etc) serán procesados en el orden en que llegan (FIFO o primero en entrar, primero en salir). Esto implica que ante la misma sucesión de eventos en el tiempo, el UE se comportará de forma determinista. Además de procesar el evento, se procesarán también las consecuencias que éste tenga en cascada sobre el resto del plan, lo que se conoce como semántica *run-to-completion* [7]. Dichos eventos serán los que conduzcan como tal el flujo de ejecución del plan.

En el punto de partida, el plan se encuentra en un estado inactivo. Esto quiere decir que todos los nodos estarán detenidos y no afectarán al mundo externo. El

primer nodo en pasar a ejecución será la raíz y, en el instante siguiente, todos sus hijos pasarán al estado de espera antes de ser ejecutados. En dicho estado, el nodo podrá entrar en ejecución cuando se verifique su condición de inicio (siendo por defecto verdadera), pasando a ejecutar las acciones definidas en su cuerpo. Una vez ejecutado, el nodo se marcará como iteración finalizada y, en base a la presencia o no de una condición de repetición, su estado final de ejecución será en espera para nodos con condición de repetición, o finalizado para nodos sin dicha condición. Hay una excepción a esta regla, los nodos lista, que tras su ejecución pasarán a un estado finalizando que indica que el nodo ha cumplido su objetivo y esta en espera de que sus nodos hijo hayan finalizado. En caso de que un nodo lista tenga condición de repetición, la lista completa podrá volver a ejecutarse cuando ésta se cumpla, pero no al revés.

Hay tres causas que marcarán la finalización de un nodo: que complete su ejecución, la presencia de fallos o eventos externos. En el primer caso, la finalización de un nodo dependerá del tipo de éste. Por ejemplo los nodos de asignación terminarán cuando se halla establecido el nuevo valor en la variable. En el caso de los nodos lista, si el nodo ha cumplido su condición de finalización pero sus nodos hijos no han terminado su ejecución, se realizará una terminación limpia de los hijos de la forma que sigue:

1. Sólo los nodos activados o en ejecución continuarán con sus procesos de forma concurrente.

2. Los hijos pendientes de verificar sus condiciones de inicio no serán ejecutados.

3. El padre esperará a que los hijos en ejecución finalicen.

En el caso de que un nodo finalice por un fallo (pre/postcondición o condición invariante que no sea satisfecha), el nodo abortará su ejecución de forma abrupta, incluyendo el caso de los nodos lista, que no finalizarán de la forma antes descrita.

Una vez finalizado un nodo, si éste tiene condición de repetición, los datos internos del nodo se reinicializarán y el nodo volverá al estado de espera. En relación a esto, hay que indicar que las condiciones del nodo, una vez activadas, no podrán ser desactivadas posteriormente salvo por un reseteo del nodo. Esto quiere decir que si en un instante dado, un nodo en espera verifica tras un evento que su condición de inicio es correcta, el nodo comenzará la ejecución, aunque en el instante siguiente dicha condición se torne falsa nuevamente.

3.3.1. Ciclos de ejecución

Los ciclos fundamentales de ejecución de un plan PLEXIL serán los *macro steps* y los *atomic steps*. Los *macro steps* serán los pasos discretos que se ejecutarán en el plan. Dichos pasos están compuestos por *micro steps* o ejecución en paralelo de una serie de *atomic steps*. Los *atomic steps* actúan sobre cada nodo individual.

Como puede verse, el nivel de ejecución inferior (refiriéndose por inferior a que esta más alejado del mundo exterior y más centrado en el estado interno del plan) es el *atomic step*, mientras que el nivel superior será el nivel de ejecución, el cual puede verse como una serie de relaciones entre los estados del mundo exterior y el estado interno del plan. La ejecución de un plan por tanto, podrá analizarse en base a las acciones llevadas a cabo por cada nivel de ejecución de forma incremental partiendo desde el nivel inferior. En los siguientes apartados se llevará a cabo el análisis del funcionamiento de cada nivel y se finalizará poniendo todos los niveles en común para ver como se comporta la ejecución completa del UE en el ciclo de ejecución.

3.3.1.1. *Atomic step*

Un paso atómico o *atomic step* define una operación interna o una transición del estado para un nodo en un instante de tiempo. El paso atómico representa una variación de un dato interno del nodo que sólo afectará directamente a éste y nunca al estado del mundo exterior. Hay un total de 37 reglas que definen las posibles transiciones de un nodo; éstas podrán verse de forma gráfica en el apéndice A. Dichas reglas no serán aplicables a todos los nodos, dado que en ciertos estados las posibles transiciones entre estados variarán en base al tipo de nodo.

3.3.1.2. *Micro step*

Un *micro step* representa la ejecución paralela de varios pasos atómicos. Los nodos se ejecutan en paralelo, de tal forma que en un único instante de tiempo puede haber uno o más nodos realizando pasos atómicos. En este nivel se llevan a cabo varias tareas: se modifican los valores de salida y estado de los nodos en ejecución y se llevan a cabo tareas de sincronización. Aquí se establecen las políticas que permitirán resolver condiciones de carrera en caso de, por ejemplo, que dos nodos intenten acceder a una misma variable para asignarle un valor. En este caso, se utilizará una regla de precedencia para evitar accesos simultáneos a recursos compartidos.

3.3.1.3. *Quiescence cycle*

El ciclo de reposo o *quiescence cycle* indica que se han realizado todos los *micro steps* posibles hasta que todos los nodos que podían ser ejecutados han alcanzado un estado estable (a la espera de un evento, finalizado, etc). Es decir, el plan está en un estado estable a la espera de nuevos eventos que permitan continuar con la ejecución de nodos que esperan por ellos o por la finalización de nodos bloqueados hasta la llegada de los mismos.

3.3.1.4. *Macro step*

Hasta este momento la ejecución del plan se lleva a cabo sólo alterando el estado interno del plan. En los *macro steps* es donde se lleva a cabo la comunicación entre el plan y los eventos producidos por el mundo exterior.

Los *macro steps* se inician con la llegada de un nuevo evento externo al sistema. Todos los nodos que estuvierán esperando por dicho evento, realizarán transiciones a su siguiente estado en paralelo a través de *micro steps*. Estas transiciones pueden provocar cambios internos que permitan transicionar a más nodos, con lo cual se llevarán a cabo más *micro steps* hasta que se finalice la secuencia de cambios en cascada. Es en ese momento cuando se alcanza el ya mencionado *quiescence cycle*. Es decir, el estado de reposo se alcanza al final de un *macro step*, el cual no es más que una secuencia de *micro steps* ocasionados por la recepción de un evento. Una vez se finaliza el paso, el sistema puede atender nuevos eventos. Como se mencionó al principio de la sección, los eventos son atendidos en orden de llegada (FIFO) y hasta que un evento no haya sido "despachado" no se comenzará a atender al siguiente.

Una particularidad importante es que los datos del mundo exterior relacionados con el plan sólo serán leídos en la primera utilización de cada *macro step*, así como a la finalización de éste. Dicho de otra manera, se asume que los cambios del mundo exterior sólo serán visibles al comienzo y al final del paso, y que una vez leídos permanecerán inalterados. Una vez leído un dato, una nueva lectura del mismo dato en el mismo *macro step* dará el mismo valor aunque éste haya cambiado en el mundo

real. Con ello se consigue una optimización en el acceso a los valores necesarios del mundo real.

También hay que indicar que la ejecución de comandos (o actuaciones sobre el mundo exterior) no las lleva a cabo el propio plan ni el UE, sino que se subrogan a un sistema de bajo nivel encargado de ejecutar las ordenes indicadas por el plan. En este sentido, un nodo de comando solicitará la ejecución de un comando determinado y podrá continuar su ejecución o esperar a que el mundo exterior genere un evento como respuesta a dicha acción. La sincronización entre el UE y el soporte de bajo nivel se llevará a cabo a través de interfaces que se verán en la sección 3.5.

3.3.1.5. Nivel de ejecución

El nivel o ciclo de ejecución puede describirse como la sucesión de *macro steps* y la lista de eventos que se producen en el mundo a lo largo del tiempo. Esto representa como evoluciona tanto el plan como el mundo desde el inicio hasta la finalización del plan. En este punto también pueden verse los efectos producidos por la ejecución de las acciones indicadas por el plan de PLEXIL y ejecutadas por el soporte de bajo nivel asociado. En la figura 3.9 puede verse un esquema con todos los niveles de ejecución.

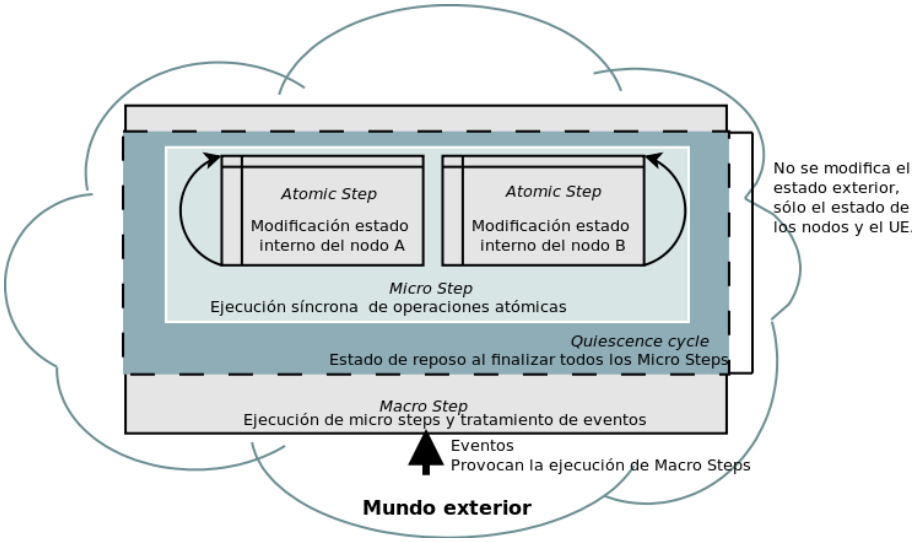

Figura 3.9: Representación de los ciclos de ejecución del UE

Finalmente hay que hacer una valoración acerca de como se trata la ejecución de PLEXIL en el tiempo. PLEXIL no hace valoraciones acerca de cuanto dura la ejecución de un paso, al contrario de lo que suelen hacer otros lenguajes. Generalmente se asume que un paso (en este caso sería un *micro step*) toma cero unidades de tiempo, o, dicho de otra forma, el mundo cambia con una frecuencia muy inferior a la duración de los pasos de ejecución. En PLEXIL esto no es importante, y se asume que si un *micro step* tarda un tiempo mayor que cero, quiere decir que el *macro step* en curso durará un tiempo mayor que cero. Durante dicho periodo, si el mundo exterior varía, puede ocurrir que los datos que utilice el plan sean obsoletos (sólo se leen los datos externos al inicio y fin del *macro step*. Esto también sucede cuando un evento se procesa mucho después de haber sido recibido. En estos casos, la lectura de los datos al final del *macro step* provocará que se actualice la información y se descarte aquella que fuera obsoleta.

3.4. Arbitrador de recursos

PLEXIL permite construir la lista de recursos necesarios para poder ejecutar un determinado comando. El arbitrador de recursos, integrado como una parte del UE, implementa la lógica necesaria para poder operar con esa información y mantener actualizada la información del uso y disponibilidad de recursos. En base a esta información, el arbitrador de recursos podrá permitir o denegar los comandos que cumplan o no con las restricciones impuestas. La figura 3.10 muestra un esquema simple de los componentes e interacciones en los que está involucrado el arbitrador de recursos.

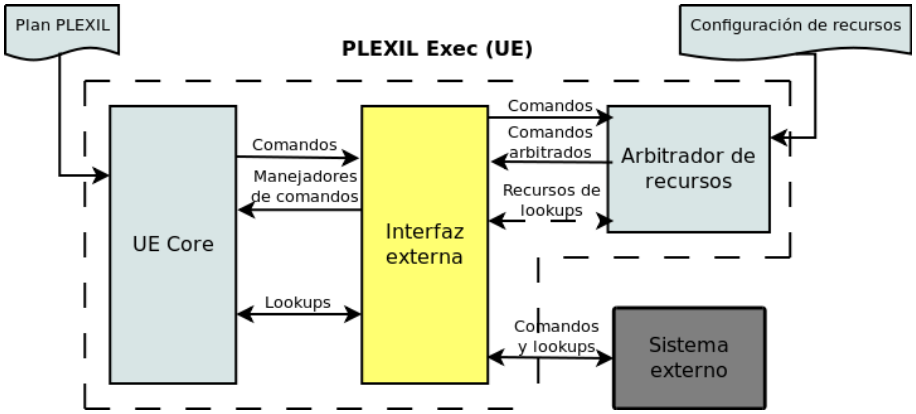

Figura 3.10: Esquema del arbitrador de recursos

El arbitrador de recursos ofrece las siguientes funcionalidades:

- Implementa recursos renovables y no renovables, que pueden ser unarios o no unarios.
- El estado de los recursos y su consumo son gestionados por él mismo. Esto asume que el consumo de cada comando es conocido con antelación y fijo. Sólo gestionará el uso de recursos de los comandos emitidos por el UE.
- No hace predicciones acerca del tiempo de duración de los comandos o del instante en que el sistema real hace uso de los recursos. Esto quiere decir que el consumo de recursos se hará inmediatamente al permitir la ejecución del comando, y, en caso de haber generación de recursos, ésta tendrá lugar al finalizar el comando.

También hay dos restricciones importantes en el arbitrador de recursos:

- Todos los recursos utilizados en un mismo comando tienen la misma prioridad (es decir, realmente no se prioriza el uso de recursos, sino que se establece prioridad al comando). No obstante, cada recurso puede especificar su propia prioridad (XML y el compilador de PLEXIL lo permiten), pero el arbitrador de recursos sólo tomará la primera prioridad establecida y la utilizará como prioridad de todos los recursos para el comando dado.
- El valor mínimo del uso de recurso es ignorado (`LowerBound`).

Es importante destacar que el arbitrador de recursos sólo se comunicará con el UE, por tanto nunca podrá hacer consultas al sistema real para conocer el estado de los recursos o comprobar la disponibilidad de los mismos. Esto responde al esquema

visto anteriormente: el grueso del sistema para la gestión de recursos es el arbitrador de recursos. Además de éste, los únicos elementos afectados por el uso de recursos es el lenguaje (se considera que el tratamiento de los recursos por parte del lenguaje es una extensión de éste) y el sistema externo, pues será el que finalmente haga uso de los recursos de que dispone. Por tanto, este sistema de control de recursos implica que en vez de enviar los comandos directamente al subsistema externo, el interfaz externo primero tiene que invocar el proceso de arbitraje y luego expedir sólo los comandos aceptados al subsistema externo.

3.4.1. Algoritmo del arbitrador

Al arbitrador de recursos le llegarán listas de comandos con sus respectivos usos de recursos desde el UE. El arbitrador entonces tendrá que gestionar los comandos para aceptar tantos comandos como sea posible comprobando que no se exceda el máximo permitido en el uso de los recursos asociados. Además, deberá tener en cuenta si hay competencia entre comandos por el uso de los mismos recursos, en cuyo caso deberá tomar en cuenta la prioridad de cada comando para el uso de recursos y aceptar sólo la ejecución del comando con mayor prioridad.

El algoritmo de funcionamiento del arbitrador de recursos es el que se expone a continuación:

1. Para que un comando sea aceptado, todos los recursos que emplea deben estar disponibles en suficiente cantidad para llevarlo a cabo.

2. El arbitrador optimiza tanto en prioridad de los recursos aceptados como en el número total de estos.

3. Un comando de menor prioridad sólo será aceptado si una vez aceptados los de mayor prioridad hay suficientes recursos como para que éste entre en ejecución. También puede darse el caso de que sea aceptado por haber denegado la ejecución a los de mayor prioridad por no cumplir las restricciones de uso de recursos.

4. Si dos recursos tienen la misma prioridad, estos serán ejecutados en el orden en el que han llegado al arbitrador al final del ciclo de espera.

5. Cuando un grupo de comandos ha sido aceptado, en el peor de los casos no se superará el uso del máximo de los recursos permitidos a cada comando.

3.4.2. Archivo de configuración de recursos

Por defecto, el arbitrador de recursos obtiene la identidad de los recursos ejecutados en el plan directamente de los comandos, cuando un comando necesita un recurso en particular, el arbitrador lo añade a su base de datos, y, una vez completada la ejecución del comando, dicho recuso será purgado de la base de datos del arbitrador. Además, establece el valor máximo para recursos, tanto renovales como consumibles, en 1.0.

No obstante, el usuario tiene la opción de recolectar la información asociada al uso de recursos del sistema a implementar y plasmarla en un archivo de configuración de recursos que podrá ser leído por el arbitrador de recursos. La expresión mínima de este archivo será una relación de los recursos usados y los valores máximos de consumo y renovación de estos. Además podrá añadir información acerca de las dependencias entre recursos. Como se mencionó anteriormente, se pueden crear jerarquías de recursos que deberán ser indicadas en este fichero, representadas

en forma de gráfico dirigido y ponderado acíclico. En la figura 3.11 se muestra un ejemplo de la estructura de una jerarquía de recursos y en la figura 3.12 el formato correspondiente para el archivo de configuración. Los pesos representan el valor absoluto del consumo del recurso indicado. En caso de no indicar peso, implica que requiere el uso del recurso en su totalidad.

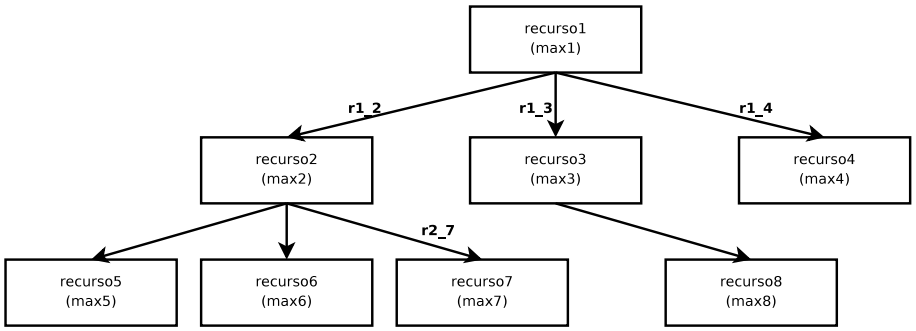

Figura 3.11: Grafo de jerarquía de recursos

```
1 recurso1 max1 r1_2 recurso2 r1_3 recurso3 r1_4 recurso4
2 recurso2 max2 1.0  recurso5 1.0  recurso6 r2_7 recurso7
3 recurso3 max3 1.0  recurso8
4 recurso4 max4
5 recurso5 max5
6 recurso6 max6
7 recurso7 max7
8 recurso8 max8
```

Figura 3.12: Archivo de configuración para el grafo de la figura 3.11

3.5. *Universal Executive Framework*

En esta sección se mostrará una guia para poder crear una aplicación que emplee el *Universal Executive* como sistema de ejecución para un plan modelado en PLEXIL. Se verá de forma básica los elementos necesarios para instanciar el UE y el prototipo básico de una librería (que utilice sólo comandos, no se verán *lookups* o la implementación de telemetría) conforme a la especificación creada para dar solución a una arquitectura típica de tres capas (3T) para la locomoción autónoma de un robot hexápodo [8].

Para poder montar una aplicación que haga uso del lenguaje PLEXIL serán necesarios dos elementos:

- **UE *framework***: será una instancia del *universal executive framework* encargada de inicializar los elementos del ejecutor, fundamentalmente los interfaces (a partir del archivo de configuración de interfaces que se verá en la sección 3.5.1) y los planes y librerías escritas en PLEXIL (el plan a ejecutar). No será necesaria la modificación del *framework* para aplicaciones sencillas; los elementos necesarios para su instanciación se verán en la sección 3.5.2.1.

- **Una o más librerías dinámicas**: en función de las directrices impuestas por el archivo de configuración de interfaces se deberán tener una o más librerías

dinámicas que contengan la implementación de los comandos y *lookups* empleados en el plan PLEXIL. La forma de crear estas librerías se vera en la sección 3.5.2.2 y requieren la especificación de una serie de funciones virtuales.

El funcionamiento de una aplicación que emplee el lenguaje PLEXIL y el ejecutor UE a la hora de realizar un comando es el que se muestra en la figura 3.13. Cuando un plan PLEXIL invoca un comando, el *framework* del UE (esto es, el ejecutor) solicita a la interfaz que lleve a cabo el comando. La interfaz del ejecutor posee un archivo de configuración definido por el usuario (archivo de configuración de interfaces) que indica que comandos se pueden llevar a cabo y que librería dinámica implementa dicho comando. Una vez localizada la librería para el comando solicitado se asignará un manejador de comando desde el ejecutor y se solicitará a la librería que lleve a cabo el código de la función asociada a dicho comando. Dentro de dicho código se podrá actualizar tanto el estado del manejador de comando, como el valor de retorno del comando si lo tuviera. La librería para poder operar de cara al ejecutor deberá poseer una interfaz específica que permita al ejecutor la solicitud de comandos de forma genérica.

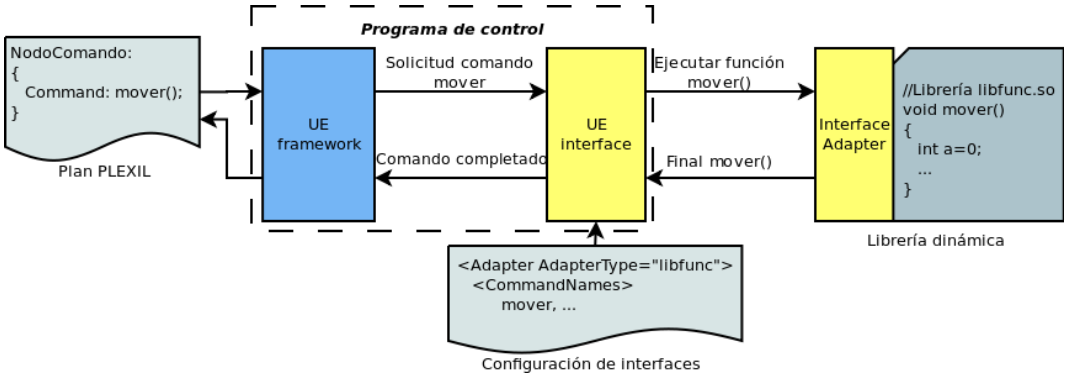

Figura 3.13: Flujo de ejecución de un comando

3.5.1. Archivo de configuración de interfaces

El archivo de configuración de interfaces será un archivo con codificación XML que indicará las librerías empleadas por el plan PLEXIL, comandos y *lookups* que lleva a cabo cada una de ellas. La etiqueta de mayor nivel del archivo será `Interfaces`. Por cada librería dinámica a emplear se deberá definir un adaptador de interfaz mediante la etiqueta `Adapter` y el atributo `AdapterType= "`*`nombre_libreria`*`"`, donde *nombre_libreria* es el nombre de la librería dinámica a emplear (sin necesidad de extensión ni el prefijo `lib` estándar).

Tras esto se podrán declarar para cada adaptador la lista de comandos o *lookups* de los cuales se encarga cada librería. Ambos casos serán una lista de los nombres de los comandos (sin lista de parámetros ni tipo de retorno si lo tuvieran) separados por comas. La etiqueta que define el listado de comandos es `CommandNames` y para los *lookups* será `LookupNames`. En el caso de que se quiera declarar una librería a la que asociar todos los comandos que no se han asociado explicitamente con otra librería, se deberá poner la etiqueta `DefaultAdapter` tras el identificador del adaptador.

En la figura 3.14 se muestra un archivo de interfaces de ejemplo. Además de las librerías que creemos para nuestra propia aplicación, la distribución de PLEXIL incorpora algunas librerías con útiles como gestión de *sockets*, utilidades variadas, gestión del tiempo del sistema operativo nativo (`OSNativeTime`) y el adaptador `Dummy` que retorna siempre COMMAND_SUCCESS.

```
<Interfaces>
   <Adapter AdapterType="OSNativeTime">
      <LookupNames>
      time
      </LookupNames>
   </Adapter>
   <Adapter AdapterType="Dummy">
      <DefaultAdapter/>
   </Adapter>
      <Adapter AdapterType="PDDLue">
      <CommandNames>
      crear_planificador, planificar, sig_accion,
      insertar_predicado, modificar_predicado,
      objeto_predicado, objeto_accion, valor_funcion,
      </CommandNames>
      <LookupNames>
      duracion
      </LookupNames>
   </Adapter>
</Interfaces>
```

Figura 3.14: Ejemplo de archivo de configuración de interfaces

3.5.2. Implementación del ejecutor

En esta sección se mostrará de forma breve mediante un ejemplo los elementos necesarios para instanciar el ejecutor UE y la creación de una librería dinámica que sea capaz de interactuar con él. El esquema básico de la aplicación es el que se muestra en la figura 3.15, en la cual se puede distinguir en la parte superior el programa principal que hace uso de una clase que instancia el ejecutor, así como una clase C++ que ha heredado la clase `InterfaceAdapter` y emplea las funciones de una librería para dar servicio al ejecutor. Además se pueden ver las dependencias directas del código de la distribución de PLEXIL que tiene cada elemento.

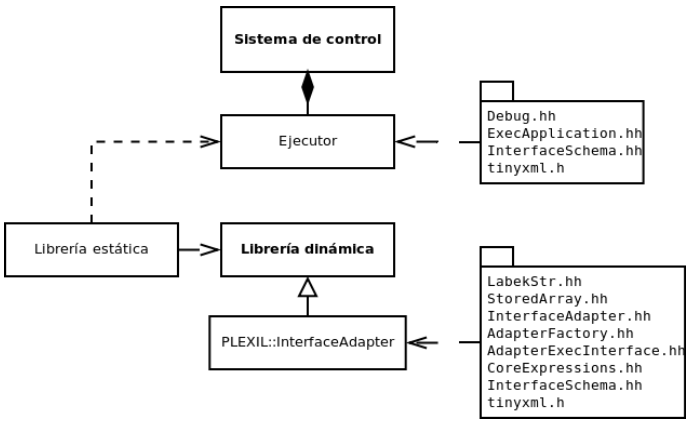

Figura 3.15: Esquema de la arquitectura empleada

Lo mostrado en las siguientes secciones será el desarrollo empleando la versión 0.90 de la distribución de PLEXIL. En versiones posteriores pueden darse cambios. La distribución de archivos dentro de una carpeta raíz (sea `src`) empleada es la siguiente:

- `main`: codigo fuente de la función principal del programa.

- Carpeta *comandos*: aquí se encuentra el código fuente para crear la librería estática `comandos`.
- Carpeta *PLEXIL*: dentro de esta carpeta se encuentran los archivos `Makefile` para crear el ejecutor y la librería dinámica. Ademas se encuentra el código asociado a los mismos (una clase por cada uno: `run_UE` y `comandosInterfaceAdapter` respectivamente) y dos subcarpetas:
  - Carpeta *tinyxml*: esta carpeta proviene de la distribución de PLEXIL y tiene los componentes necesarios del parseador XML.
  - Carpeta *src*: aquí se encuentran los componentes de PLEXIL. Estará compuesta por las carpetas *app-framework*, *exec* y *utils* que se pueden encontrar en `$PLEXIL_HOME/src`.

3.5.2.1. Implementación del *framework* del UE

Para poder compilar los elementos necesarios del *framework* en la carpeta *app-framework* se encuentra un archivo `Makefile` que podemos adaptar a nuestras necesidades con el fin de compilar el sistema de ejecución de nuestra aplicación.

En este ejemplo en concreto tendremos una clase que instanciará los elementos del ejecutor `run_UE` y una clase `main` que se encargará de crear el resto del contenido de nuestra aplicación. Además se hará uso de una librería dinámica (`comandos`), que es en origen el código de una librería estática. El `Makefile` empleado es el que se muestra en la figura 3.16[2].

```
1  include makeinclude/standard-defs.make
2  EXECUTABLE  := a.out
3
4  # External dependencies
5  INC_DIRS += tinyxml src/utils src/exec src/app-framework ../
       comandos ../../include
6  LIB_PATH += ../../lib
7  LIBS     += tinyxml PlexilUtils PlexilExec PlexilAppFramework
        Comandos dl
8  SRC      := ../main.cpp run_UE.cpp
9  INC      :=
10
11 include makeinclude/standard-targets.make
12 ifneq (\$(MAKECMDGOALS),clean)
13 -include Makedepend
14 endif
```

Figura 3.16: Ejemplo de `Makefile` para el ejecutor

El archivo `run_UE` es básicamente una copia del archivo `$PLEXIL_HOME/src/app-framework/test/testMain.cc` el cual permite instanciar mediante el parámetros (para definir el plan PLEXIL, librerías PLEXIL, archivo de configuración de interfaces y archivo de depuración) el ejecutor y ejecutar el plan PLEXIL indicado.

3.5.2.2. Implementación de una librería dinámica para el UE

Para utilizar una librería en un plan PLEXIL se deberá implementar la clase C++ `InterfaceAdapter` de forma que de soporte a nuestras necesidades. Dicha

[2]Se ha omitido la licencia contenida en el `Makefile` original.

clase posee una serie de funciones virtuales para permitir codificar el funcionamiento de los comandos y *lookups*. El funcionamiento de ambos es similar, de tal forma que la ejecución de un comando en un plan PLEXIL implica una llamada a la función `executeCommand` con una serie de parámetros que permiten conocer el nombre del comando, los parámetros utilizados al invocar el comando y el manejador asociado a éste. Con ello se podrán realizar las llamadas pertinentes a las funciones implementadas en el sistema externo, de tal forma que el resultado obtenido por dichas funciones podrá ser devuelto al ejecutor a través del manejador de comando asociado, mediante la función `handleValueChange`. Además con la misma función se puede asociar al manejador el estado de ejecución del comando en cualquier momento durante la ejecución del mismo. Dicho estado acepta los valores vistos en la sección 2.11.2.

Al igual que en el caso anterior, dispondremos de un `Makefile` genérico que podremos adaptar para crear nuestra propia librería. El archivo empleado para este ejemplo es el que se muestra en la figura 3.17. Con ello obtendremos la librería `libComandos.so` que podremos emplear para dar servicios a los planes PLEXIL que utilicen los comandos implementados.

```
1  include makeinclude/standard-defs.make
2  OSTYPE    ?= $(shell uname -s)
3  MACHTYPE ?= $(shell uname -p)
4  OS_ARCH = $(OSTYPE)-$(MACHTYPE)
5
6  LIBRARY := Comandos
7  LIBRARY_SRC := comandosInterfaceAdapter.cpp
8  LIBRARY_OBJ := $(LIBRARY_SRC:.cpp=.o)
9  INC_DIRS += tinyxml src/utils src/exec src/app-framework ../
       comandos ../../include
10 LIBS += tinyxml Comandos
11 LIB_TARGET = lib$(LIB_NAME)$(LIB_EXT)
12
13 include makeinclude/standard-targets.make
14 # these must follow the above 'include'
15 $(SHLIB): depend $(LIBRARY_OBJ)
16     $(LD) $(SHARED_FLAGS) $(LIB_PATH_FLAGS) $(LIB_FLAGS) -o $@
           $(LIBRARY_OBJ)
17 $(ARCHIVE): $(LIBRARY_OBJ) depend
18     $(AR) -o $@ $(LIBRARY_OBJ)
19 -include Makedepend
```

Figura 3.17: Ejemplo de `Makefile` para una librería dinámica

El archivo `comandosInterfaceAdapter` será una clase C++ que hereda de la clase `PLEXIL::InterfaceAdapter`. Con ello dispondremos de una serie de funciones virtuales que deberemos implementar para dar soporte a la interfaz con el ejecutor. En este ejemplo sólo se verá el funcionamiento de los comandos. Se verá cómo se reciben los comandos, el parseo de parámetros, modificación del estado del manejador de comando y el retorno de un valor como resultado de la ejecución del comando.

Antes de crear cualquier otro elemento de código habrá que registrar el adaptador como se muestra en la figura 3.18.

Una vez hecho esto podremos pasar a implementar las funciones necesarias para que la librería cumpla su cometido. Aquí sólo se verán los elementos básicos, pero se puede definir el comportamiento deseado en función de las necesidades de la librería y la aplicación para, por ejemplo, la inicialización de la librería, finalización, reinicio de la misma, etc. Si deseamos que la librería dispon-

```
1 extern "C" {
2    void initcomandos() {
3       REGISTER_ADAPTER(comandosInterfaceAdapter,"Comandos");
4    }
5 }
```

Figura 3.18: Registro del adaptador de una librería

ga de datos iniciales o si tiene que crear una estructura de datos en su inicialización, se podrá incluir el código necesario en la función `initialize()`. Además en dicha función habrá que iniciar un atributo heredado que permitirá conectar la librería con la interfaz del ejecutor. Para ello habrá que emplear la instrucción `m_execInterface.defaultRegisterAdapter(getId());`. El resto de funciones (`start()`, `stop()` o `reset()` por ejemplo) se pueden dejar con un simple `return true;` si no se requiere que tengan un comportamiento específico.

Las funciones que dan soporte a la interfaz del ejecutor son `lookupNow()`, `lookupOnChange()` y `executeCommand()`. Las primeras se encargan de las solicitudes de *lookups* y la última, en la que entraremos en más detalle, de los comandos.

A la hora de recibir un comando, el punto de entrada a nuestro código será la función `executeCommand` que recibirá una cadena de texto (en formato `PLEXIL::LabelStr`, consulte la documentación de PLEXIL para familiarizarse con dicha clase) que contendrá el nombre del comando a ejecutar, así como la lista de argumentos con la que ha sido invocado, la variable destino para el valor de retorno del comando y el manejador del comando que permitirá notificar al ejecutor en que estado de ejecución se encuentra nuestro comando en cada momento.

Una vez tengamos la identificación del comando solicitado deberemos proceder a ejecutar el código pertinente. Una de las primeras cuestiones será notificar al ejecutor que el comando ha sido recibido, para ello se empleará la función `handleValueChange()`, siendo los parámetros el manejador de comando y la variable que determina el nuevo estado del manejador (pueden verse los estados posibles en la sección 2.11.2). Una vez modificado el valor del manejador se deberá notificar el cambio al ejecutor para que éste modifique su estado interno. Ello se hace con la función `notifyOfExternalEvent()`. Es importante mencionar que estas funciones operan directamente sobre el objeto `m_execInterface`. En cualquier momento se podrá modificar el estado del manejador del comando, así como notificar el valor de retorno del comando. Esto también se hará con la función `handleValueChange()` pero se empleará como parámetro la variable destino indicada como parámetro en la función de entrada. Hay que tener en cuenta que todos los valores notificados al ejecutor deberán de ser de tipo *double*, de tal forma que las cadenas de caracteres deberán ser convertidas a objetos `LabelStr` que están definidos por un identificador de tipo *double*, o se deberán emplear los objetos de la clase `BooleanVariable` para definir los tipos booleanos. Una vez actualizado el valor de la variable no será necesario notificar al ejecutor de dicho cambio.

Finalmente, una vez ejecutado el código de nuestra función, antes de que finalice la función `executeCommand()` se deberá modificar el resultado del comando (COMMAND_SUCCESS o COMMAND_FAILED) y notificar el cambio en el manejador al ejecutor como se ha visto anteriormente. En la figura 3.19 se puede ver un ejemplo para una función `executeCommand()` que da servicio a dos comandos.

```
1  void PDDLInterfaceAdapter::executeCommand(const PLEXIL::
       LabelStr& name, const std::list<double>& args, PLEXIL::
       ExpressionId dest, PLEXIL::ExpressionId ack)
2  {
3      bool success = true;
4      std::string nStr = name.toString();
5      std::list<double>::const_iterator largs = args.begin();
6      int nplanner = static_cast<int> (*largs);
7      m_execInterface.handleValueChange(ack, PLEXIL::
           CommandHandleVariable::COMMAND_RCVD_BY_SYSTEM());
8      m_execInterface.notifyOfExternalEvent();
9      if(nStr == "comando1")
10     {
11         PLEXIL::LabelStr arg1(*largs);
12         printf("Argumento: \%s",(char*)arg1.c_str());
13         ultimaacc.push_back(new stpredicado);
14         unsigned int num = 1;
15         m_execInterface.handleValueChange(dest,num);
16     }
17     else if(nStr == "comando2")
18     {
19         bool arg2 = static_cast<bool> (*(++largs));
20         if(!arg2)
21            success=false;
22         m_execInterface.handleValueChange(dest,success?PLEXIL
               ::BooleanVariable::TRUE():PLEXIL::BooleanVariable
               ::FALSE());
23     }
24
25     if(success)
26         m_execInterface.handleValueChange(ack, PLEXIL::
               CommandHandleVariable::COMMAND_SUCCESS());
27     else
28         m_execInterface.handleValueChange(ack, PLEXIL::
               CommandHandleVariable::COMMAND_FAILED());
29     m_execInterface.notifyOfExternalEvent();
30 }
```

Figura 3.19: Ejemplo de función `executeCommand()` para dos comandos

3.5.2.3. Implementación de una librería dinámica para el UE a partir de SampleAdapter

Para la implementación de una librería dinámica para el UE, se propone la edición y adaptación por parte del programador de un entorno desarrollado por la *Universities Space Research Association* (USRA), cuyo código se puede encontrar dentro del repositorio de PLEXIL. Dentro de los archivos que se necesitan para la implementación, encontramos dos archivos fuente (y sus correspondientes *headers*) `types.cc` y `subscriber.cc` que serán comunes a todas las interfaces a desarrollar, así como otros dos archivos fuente específicos, `SampleAdapter.cc` y `sample_system.cc`.

Internamente, PLEXIL trata todos los tipos de datos como `double`, pese a su definición en `Int`, `Real`, `String`, `Bool` y `Array`. El primer archivo fuente, `types.cc` y `types.hh` crea unas funciones muy útiles para poder transformar todos los tipos de datos que se utilicen en la librería, a los datos utilizados en PLEXIL. Destacando un tipo de dato `Any` que se utilizará especialmente para la gestión de los estados y *lookups* de la librería. Esta abstracción que aporta este archivo, facilita el trabajo para evitar el empleo de operadores *cast*. Estas funciones son del tipo `encodeTYPE` y `decodeTYPE`, siendo `TYPE` el tipo de dato que admite PLEXIL.

Por otra parte, `subscriber.cc` y `subscriber.hh` implementan funciones para la actualización de los valores de los estados utilizados en la librería a implementar. Por su forma de implementación mediante sobrecarga del método `publish`, se hace más sencillo la modificación de un estado, sin atender al tipo de dato de estado, parámetros necesarios, etc.

El primero de los archivos a modificar es `SampleAdapter.cc`. Se propone no añadir métodos a la clase directamente en este archivo fuente, dado que ese es el objetivo de `sample_system`. Por una parte, debemos implementar las operaciones que deseemos en el `executeCommand`, que serán ejecutadas según el plan de PLEXIL creado, así como el control de los *lookups* en el método `lookupNow` que se explica a continuación.

Cuando el UE encuentra un nodo del tipo *Command node*, llama al método `executeCommand` indicándole el nombre del comando, mediante `PLEXIL::LabelStr`, una lista de argumentos del tipo `Any` visto anteriormente, y los argumentos `dest` y `ack` del tipo `PLEXIL::ExpressionId` utilizados por el UE para el manejador de comandos, que indicara si ha sido ejecutado correctamente, y la valor de retorno de la función. El funcionamiento de este método es simple, compara el nombre del comando a ejecutar (`PLEXIL::LabelStr command_name`) con los valores (*string* o `PLEXIL::LabelStr`) de aquellos comandos a implementar, y llama a la función correspondiente. En caso de no coincidir con ningún comando, se mostrará un error.

Por último, en caso de devolver un valor, se pasará mediante el manejador. Los argumentos del comando, son pasados mediante una lista de objetos tipo `Any`, que serán convertidos a tipos de datos de PLEXIL, mediante el uso de las funciones `encodeTYPE` y `decodeTYPE` vistas anteriormente.

Como se puede apreciar en la figura 3.20, el uso tan sólo de llamadas a las funciones de ejemplo `setValorX`, `avanzaRover` y `calculaAltura` simplifica la sintaxis del código. Estas funciones, son funciones externas al presente código, y a modo de ejemplo, se implementarán en el archivo `sample_system.cc`.

Durante el desarrollo de este documento, se ha explicado el significado y el uso de los *lookups* en el código, que atienden a modificaciones externas del estado del sistema. En caso de alteración del valor de un estado, el ejecutor llama al método `lookupNow` a continuación (figura 3.21). Este método, según se desarrolla en el ejemplo, toma el nombre del estado modificado, así como de los argumentos de este para su actualización de información. Dependiendo de la aplicación que se desee desarrollar, no es necesario la modificación del código perteneciente a `lookupNow`, sí en cambio la función `fetch`.

La función `fetch`, de tipo `Any` hace una selección del estado que se desea modificar, entre los que se quieran tener disponibles, de una forma muy similar a lo visto en `executeCommand`, y se pasan los argumentos por referencia, mediante las funciones (`getX` y `setX`) creadas en el archivo `sample_system.cc`.

Con todo lo visto anteriormente, el archivo `sample_system.cc` deberá contener los prototipos y desarrollos de las funciones que son accedidas desde el `executeCommand`, así como la definición de los estados y sus funciones `setX` y `getX`. Este archivo es, principalmente, el que va a llevar toda la carga de funcionalidad para procesar los datos desde los `Command nodes` de PLEXIL. Por conveniencia, se propone el uso de diferentes archivos con estas funcionalidades, para las distintas librerías que se deseen crear, a fin de agilizar y depurar con facilidad, pero no hay ninguna regla que lo impida. A continuación, se muestra un ejemplo con las diferentes partes que componen este código de ejemplo. En la figura 3.23 se definen los estados del sistema como variables de tipo `static`, que pueden o no ser inicializadas. En la figura 3.24, mediante la función `defAccessors` se generan automáticamente las funciones `getX` y `setX`. Por último, la figura 3.25 contiene un pseudoejemplo de un prototipo de

```
1  void EjemploAdapter::executeCommand (const LabelStr&
       command_name, const list<Any>& args, PLEXIL::ExpressionId
       dest, PLEXIL::ExpressionId ack)
2  {
3     // name contiene el nombre del comando ejecutado por el UE
4    string name = command_name.toString();
5    debugMsg("EjemploAdapter", "executeCommand recibido " <<
         name);
6
7     // Variable para el resultado, en caso de necesitar return
8    Any resultado = Unknown;
9
10    // Lista de argumentos del comando
11   vector<Any> argv(10);
12   copy (args.begin(), args.end(), argv.begin());
13
14   if (name == "Funcion1")
15   {
16    // Este comando no devuelve valor, pasa un único argumento
          (Real) a la función
17    setValorX (decodeReal (*args.begin()));
18   } else if (name == "Funcion2")
19   {
20    // Este comando no devuelve valor, pasa dos argumentos
21    avanzaRover (decodeReal(argv[0]), decodeReal(argv[1]),
          decodeInt (argv[2]));
22   } else if (name == "CalculaAltura")
23   {
24    // Este comando devuelve el valor retornado por la función
          calculaAltura
25    resultado = calculaAltura (decodeInt (*args.begin()));
26   } else cerr << error << "Comando no valido: " << name <<
         endl;
27
28   // Manda un renorno al manejador de comandos del UE
29   m_execInterface.handleValueChange
30     (ack, PLEXIL::CommandHandleVariable::
          COMMAND_SENT_TO_SYSTEM());
31
32   // Devuelve el resultado del comando, si procede
33   if (dest != PLEXIL::ExpressionId::noId()) {
34     m_execInterface.handleValueChange (dest, resultado);
35   }
36
37   m_execInterface.notifyOfExternalEvent();
38 }
```

Figura 3.20: Ejemplo de función `executeCommand()` basado en SampleAdapter.cc

función que es llamada desde `executeCommand`, donde se deberá implementar toda la funcionalidad que se necesite para la ejecución del nodo de PLEXIL.

```
double EjemploAdapter::lookupNow (const State& state)
{
  // Nombre del estado que se vaya a modificar, y sus
      argumentos
  LabelStr name (state.first);
  const vector<Any>& args = state.second;
  return fetch(name.toString(), args);
}
```

Figura 3.21: Ejemplo de función `lookupNow()` basado en SampleAdapter.cc

```
static Any fetch (const string& state_name,
                  const vector<Any>& args)
{
  debugMsg("EjemploAdapter:fetch",
           "Llamada a Fetch de  " << state_name << ",
               argumentos ( " << args.size() << " ) " );

  Any resultado;

  if (state_name == "Estado1")
  {
   resultado = encodeReal (getEstado1(decodeReal(args[0])));
  } ...
   else {
    cerr << error << " estado no valido: " << state_name <<
        endl;
    resultado = Unknown;
  }
  debugMsg("EjemploAdapter:fetch",
           "Valor retornado por Fetch: " <<
           PLEXIL::Expression::valueToString (resultado));
  return resultado;
}
```

Figura 3.22: Función `fetch()` basado en SampleAdapter.cc

```
// Estados del sistema, como variables.
//
static float Estado1 = 23.4;
static int Estado2 = 7;
static string Color = "Azul";
static book Encendido = true;
static pair<int, int> Coordenadas (0,0);
```

Figura 3.23: Declaración de estados en `sample_system.cc`

```
1  // Función para la generación de las funciones getX y setX
2  // Se pasa como referencia name (nombre del estado) y el tipo
       de dato
3  // que corresponda en cada caso
4
5  #define defAccessors(name, type) \
6  type get##name () \
7  { \
8    return name; \
9  } \
10 void set##name (const type & s) \
11 { \
12   if (s != name) { \
13     name = s; \
14     publish (#name, s); \
15   } \
16 }
17
18 // Llamada para la generación de los funciones que se
       necesiten
19 defAccessors(Estado1, float)
20 defAccessors(Estado2, int)
21 defAccessors(Color, string)
22 defAccessors(Encendido, bool)
```

Figura 3.24: Funciones `getX` y `setX` en `sample_system.cc`

```
1  void avanzaRover (static double coorX, static double coorY,
       static double Pasos)
2  {
3    static double x, y, p;
4    if (coorX > 0) {
5      ...
6    }
7  }
```

Figura 3.25: Funciones Command en `sample_system.cc`

Apéndice A

Transiciones de los nodos en PLEXIL

En este anexo se presentan la posibles transiciones entre los diferentes estados que podrán llevar a cabo los nodos en PLEXIL. Para ello, se dará una breve explicación anexa a una figura que representará las posibilidades en función del estado inicial y el tipo de nodo.

En la figura A.1 se muestra la leyenda para las imágenes de este anexo[1]. Los rectángulos amarillos representan las posibles condiciones que pueden hacer que un nodo cambie de estado. Los rectángulos con barras oscuras a los lados indican el estado de salida de un nodo. Los rombos lilas implican una comprobación de condición. La comprobación de una condición puede darse para la lógica bivaluada (verdadero o falso), o para la trivaluada (verdadero, falso o desconocido). En las figuras se representarán con T, F y U para verdadero, falso y desconocido respectivamente (*True, False and Unknown*). Para indicar las transiciones se utilizan flechas desde el estado origen al estado destino. En caso de haber múltiples transiciones posibles, un número entero representará el orden de preferencia, siendo el menor valor el más prioritario. Finalmente, las elipses azules indican en su interior el estado en el cual se encuentra el nodo.

[1]Todas las imagenes de este apéndice han sido obtenidas de la wiki de Plexil: `http://sourceforge.net/apps/mediawiki/plexil/index.php?title=Node_State_Transition_Diagrams`

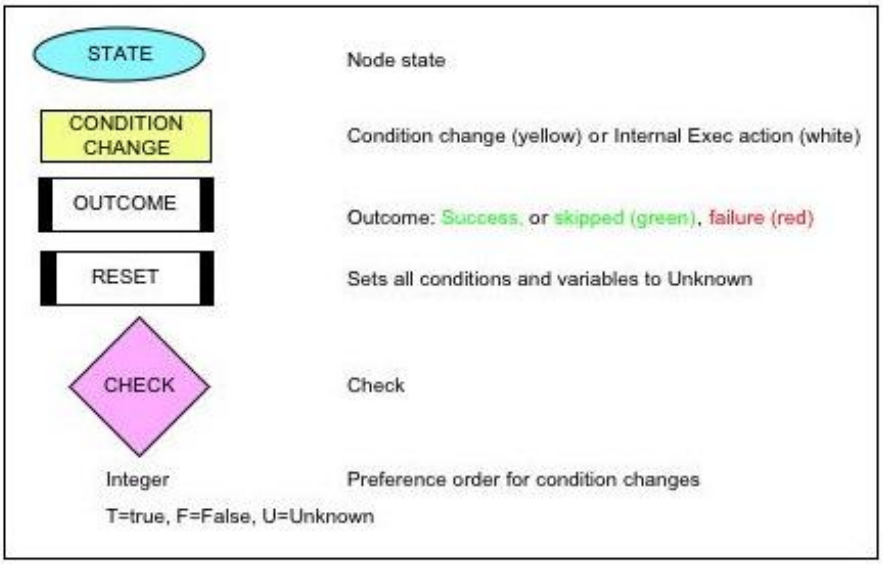

Figura A.1: Leyenda

En la figura A.2 se representa el punto de partida para todos los nodos. Todos los nodos comienzan en estado inactivo y pueden variar a omitido o en espera en base las condiciones de ejecución de su nodo padre.

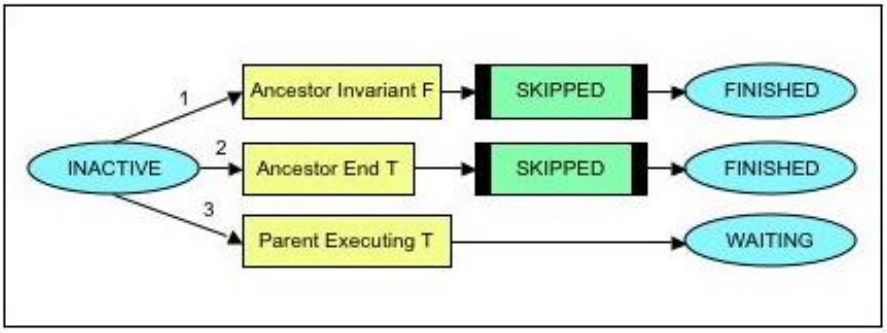

Figura A.2: Transiciones desde “inactivo” para todos los tipos de nodos

En la figura A.3 se muestran las transiciones posibles una vez que el nodo esta preparado para poder entrar en ejecución. El nodo podrá finalizar como omitido si se dan ciertas condiciones en el padre, o, en caso de no darse estas, comprobar sus condiciones de inicio pudiendo ser apto para ejecución o finalizar en fallo por no haber satisfecho dichas condiciones.

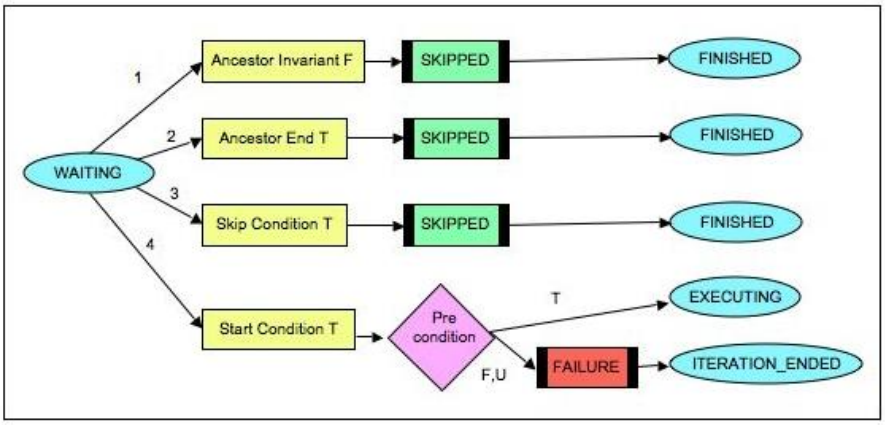

Figura A.3: Transiciones desde "en espera" para todos los tipos de nodos

La figura A.4 se presenta como finaliza un nodo de tipo lista. Hay que recordar que estos nodos sólo comprobarán su condición de finalización una vez todos los nodos hijos hayan finalizado su ejecución.

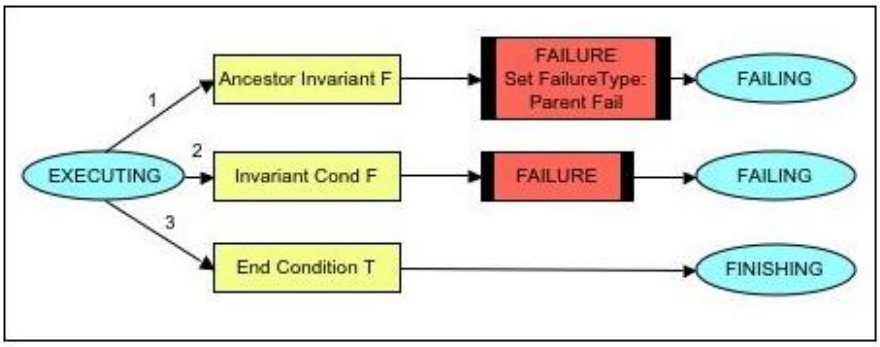

Figura A.4: Transiciones desde "en ejecución" para los nodos de tipo lista

La figura A.5 muestra las posibles transiciones desde el estado de ejecución al de finalización para los nodos de tipo comando y actualización. En estos nodos entra en juego la condición invariante que puede provocar una finalización prematura del nodo. Esta terminación abrupta implicará, en caso de no haber cumplido el objetivo del nodo, abortar la ejecución de las tareas del nodo. El estado final *"ITERATION_ENDED"* implica que el nodo ha finalizado un ciclo de ejecución y que esta listo para volver a ser ejecutado si existe y se cumple la condición de repetición.

En la figura A.6 pueden verse las posibles transiciones en la ejecución de un nodo de asignación. Estos nodos varían de los anteriores en la forma de abortar la ejecución del nodo ante un fallo. En este caso asignarán el valor desconocido a la variable que tuvieran que tratar antes de finalizar la ejecución. La condición de

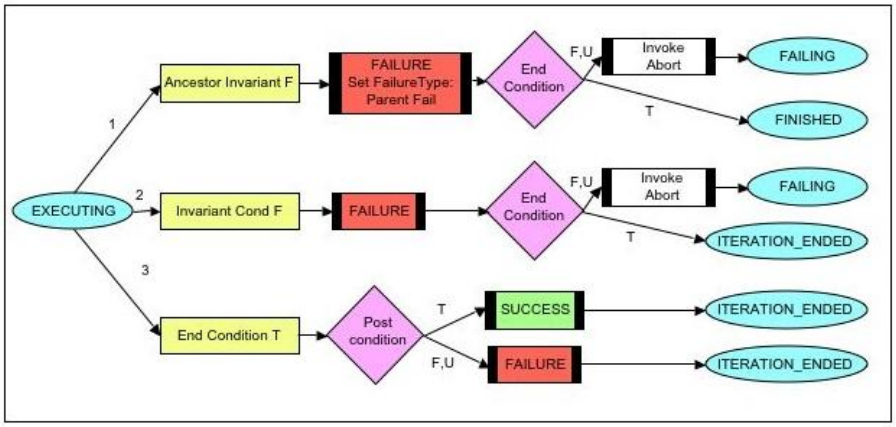

Figura A.5: Transiciones desde "en ejecución" para los nodos de comando y actualización

finalización puede utilizarse para evaluar la corrección de la asignación mediante la lectura del valor de salida.

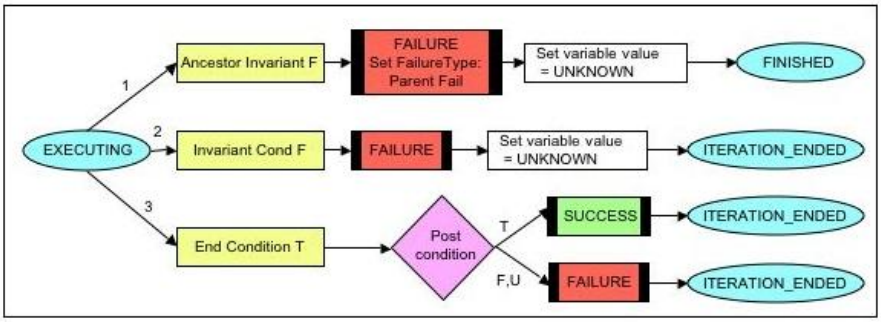

Figura A.6: Transiciones desde "en ejecución" para los nodos de asignación

En la figura A.7 se muestra el comportamiento de los nodos vacíos. Dado que el uso de estos nodos es generalmente comprobar una condición para leer su estado de salida, su comportamiento más común es comprobar su condición de finalización y establecer el valor de salida en consecuencia para que otro nodo lo utilice.

En base al estado de ejecución de los nodos hijos y del estado del salida del nodo padre, un nodo de tipo lista establecerá la causa de su fallo en su valor de salida. Dicha eventualidad puede verse en la figura A.8.

Al igual que en el caso anterior, los nodos de tipo comando y actualización establecerán el tipo de fallo ocurrido durante su ejecución en base a si el fallo es propio o del padre, siempre y cuando la ejecución del nodo haya sido correctamente abortada. El flujo puede verse en la figura A.9.

La figura A.10 representa las transiciones posibles para la finalización de un nodo de tipo lista. Para que estos nodos terminen correctamente, todos sus hijos deberán haber finalizado su ejecución (independientemente de como haya sido esta) y deberá verificar su condición de finalización.

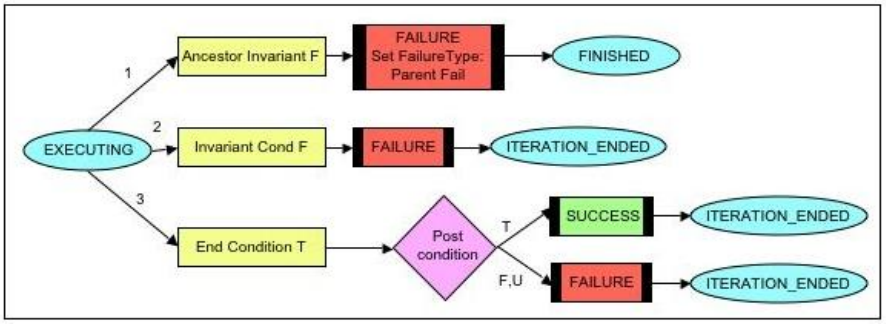

Figura A.7: Transiciones desde "en ejecución" para los nodos vacios

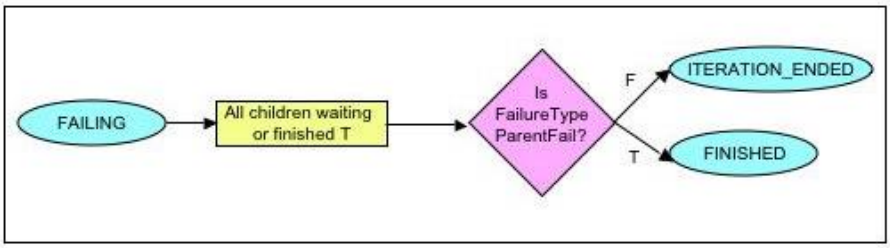

Figura A.8: Transiciones desde "fallo" para los nodos de tipo lista

En la figura A.11 se muestran los posibles flujos de ejecución para todos los tipos de nodos desde el estado de *"ITERATION_ENDED"*. En caso de que el padre haya finalizado su ejecución, el nodo finalizará también la suya, independientemente de que tenga o no condición de repetición. Si el nodo padre todavía no ha finalizado y el nodo tiene condición de repetición, si esta se cumpliese, podría volver a ejecución posteriormente, modificando su estado a "en espera" o, en caso contrario, finalizar su ejecución.

La figura A.12 muestra como uno nodo finaliza completamente su ejecución. Una vez haya finalizado, esperará a que su padre también lo haga, en cuyo momento el nodo borrará todos sus datos internos y pasará al estado "inactivo ".

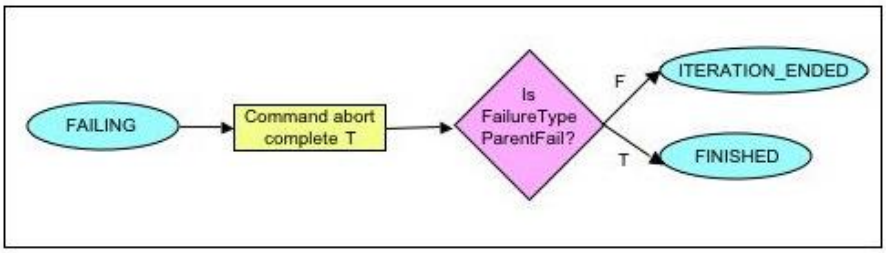

Figura A.9: Transiciones desde "fallo" para los nodos de comando y actualización

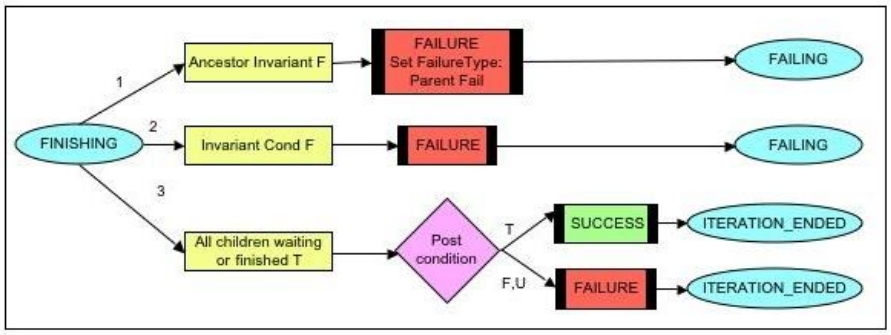

Figura A.10: Transiciones desde "finalizando" para los nodos de tipo lista

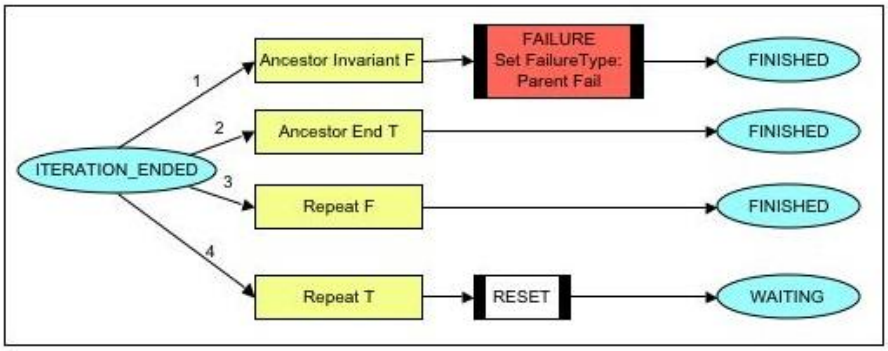

Figura A.11: Transiciones desde "*ITERATION_ENDED*" para todos los tipos de nodos

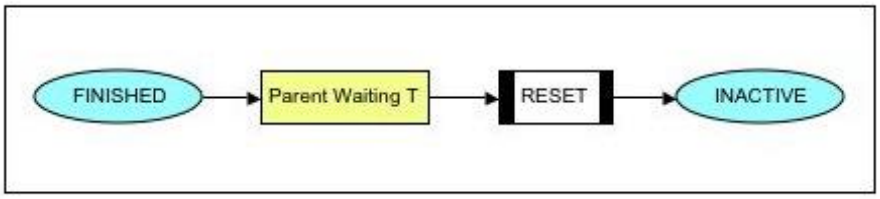

Figura A.12: Transiciones desde "finalizado" para todos los tipos de nodos

Referencias

[1] Universities Space Research Association (USRA). PLEXIL and the Universal Executive. [Online]. Available: http://sourceforge.net/projects/plexil/

[2] M. Bualat, L. Edwards, T. Fong, M. Broxton, L. Flueckiger, S. Lee, E. Park, V. To, H. Utz, V. Verma, C. Kunz, and M. MacMahon, "Autonomous robotic inspection for lunar surface operations," *Field and Service Robotics Conference*, 2007.

[3] USRA, "PLEXIL and Universal Executive Reference Manual," Tech. Rep., October 2010.

[4] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version control with Subversion.* O'Reilly, 2004, [Online]. Available: http://svnbook.red-bean.com/.

[5] D. McDermott, "The PDDL Planning Domain Definition Language," *The AIPS-98 Planning Competition Comitee*, 1998.

[6] W3 Consortium. Date and Time Formats. [Online]. Available: http://www.w3.org/TR/NOTE-datetime/

[7] G. Dowek, C. Muñoz, and C. S. Pasareanu, "Formal semantics of a synchronous plan execution language," *Workshop on Planning and Plan Execution for Real-World Systems: Principles and Practices for Planning in Execution at the International Conference on Automated Planning and Scheduling (ICAPS)*, 2007.

[8] P. Muñoz, M. D. R-Moreno, and B. Castaño, "Integrating a PDDL-based planner and a PLEXIL-executor into the Ptinto robot," in *23rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems: Next-Generation Applied Intelligence (IEA-AIE 2010)*, ser. Lecture Notes In Artificial Intelligence, N. G. et al., Ed. Córdoba, Spain: Springer-Verlag, june 2010, pp. 72–81.