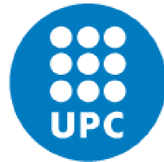
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

# CONTROL DE SISTEMES MITJANÇANT ARBRES DE COMPORTAMENT

Xavier Vales i Abenoza

Febrer de 2015

TESI DE MÀSTER
Màster en Enginyeria Informàtica
Facultat d'informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) – Barcelona-Tech

Director: Manel Frigola Bourlon
Departament: Enginyeria de Sistemes, Automàtica i Informàtica Industrial

# Índex

1. Introducció ..... 3
2. Introducció als arbres de comportament ..... 4
3. Especificació ..... 7
4. Disseny ..... 8
   4.1. Interfície gràfica ..... 8
   4.2. Programació ..... 9
   4.3. Generador de codi ..... 10
   4.4. Compilació execució ..... 11
5. Implementació ..... 12
   5.1. Implementació de l'arbre de comportament ..... 12
   5.2. Implementació de l'entorn de desenvolupament ..... 21
      5.2.1. Finestra principal. Àrea de disseny. ..... 21
      5.2.2. Programació de l'arbre de comportament ..... 24
      5.2.3. Generador del codi font a partir de l'arbre de comportament ..... 28
      5.2.4. Compilació del codi font ..... 30
      5.2.5. Exportar el projecte ..... 32
6. Test, proves, resultats i valoracions ..... 35
7. Cronograma ..... 38
8. Valoració econòmica del projecte ..... 39
9. Conclusions ..... 41
10. Treball futur i millores ..... 42
11. Annexos ..... 43
12. Bibliografia ..... 46

# 1. Introducció

Especificar de forma precisa i entenedora missions d'una certa complexitat és un dels camps d'estudi en el desenvolupament de software destinat a la indústria del videojoc, del control i de l'automatització de sistemes.

Algunes de les tècniques per realitzar el disseny d'aquest tipus de programari són la utilització d'esquemes, seqüències o grafs d'estats. Actualment, els esquemes més utilitzats són els diagrames d'estats(*Statecharts*) o els arbres de comportament. Els diagrames d'estats són grafs dirigits en que els nodes representen tots els possibles estats d'un sistema i les arestes transicions i condicions que activen aquests canvis d'estat. Els arbres de comportament, com els diagrames d'estats, són models gràfics basats en màquines d'estats finits per definir com els sistemes es comporten. És diferencien però amb els *statecharts* en el fet que els arbres de comportament s'estructuren en tasques activades segons l'estat en comptes d'estats i transicions explícites en el diagrama.

L'objectiu d'aquests esquemes és especificar el disseny del sistema de la forma més clara i precisa possible per tal de poder transformar l'especificació mitjançant un entorn IDE en codi (software) minimitzant així els errors.

Dissenyar un comportament i validar-lo pot ser complicat usant màquines d'estat finit degut al baix nivell amb què treballen i seva baixa escalabilitat. Quan el nombre d'estats creix, fet que succeeix en tasques complexes, es multiplica el nombre d'arestes, fent que sigui difícil d'entendre i assimilar els possibles estats, modificar, trobar errors i validar-ho.

Aquest projecte proposa els arbres de comportament com a eina per dissenyar software de control per microcontroladors.

Específicament, es vol crear un entorn de desenvolupament per tal que el programador pugui dissenyar i implementar tasques mitjançant arbres de comportament. Aquest entorn permetrà programar cada funcionalitat fent clic directament sobre l'estructura d'arbre que defineix el disseny, i generar posteriorment el codi de programa executable de forma automàtica.

Així doncs l'objectiu d'aquest projecte és desenvolupar un editor gràfic que permeti dissenyar de manera també gràfica i intuïtiva una tasca relativament complexa. Tanmateix es desitja que estigui també integrada l'edició i generació de codi (IDE).

La finalitat del projecte és permetre que l'usuari es pugui centrar en l'especificació de la tasca realitzant el disseny amb un editor gràfic entenedor i guiant el procés de generació de codi.

## 2. Introducció als arbres de comportament

Els arbres de comportament són una eina que es basa en una sèrie de tècniques usades en intel·ligència artificial; màquines d'estat (*Statecharts*), programació, planificació i execució de tasques[i]. A diferència de les màquines d'estats jeràrquiques que es basen en estats, els arbres de comportament es basen en tasques i els estats es troben de forma implícita en l'estructura de l'arbre.

Quan l'arbre es representa en un diagrama, el comportament s'inicia per una tasca que és l'arrel. L'arbre només està format per tasques (nodes) i enllaços als seus fills. Una tasca pot estar composta d'altres subtasques per realitzar accions més complexes.

Cada tasca s'activa per ser executada seguint l'arbre de dalt a baix (des de l'arrel) i d'esquerra a dreta. Les tasques poden tenir diversos estats interns; poden estar en execució, haver fallat, haver tingut èxit o un error. Totes les tasques es comuniquen amb el seu pare retornant el valor corresponent a l'estat de la seva pròpia execució. D'aquesta manera es propaga i controla el flux de l'execució del comportament.

Els principals tipus de tasques d'un arbre de comportament són els següents:

- **Selector**. És una tasca que activa cada una de les subtasques que té retornant èxit immediatament si qualsevol d'elles retorna èxit. El seu comportament correspon a la funció lògica OR. Es sol usar el símbol '?' per denotar un selector.
- **Seqüència**. Activa cada una de les seves subtasques en ordre, una darrera l'altra i només retorna èxit si tots els fills acaben retornant èxit. Si fracassa una subtasca retornarà fracàs immediatament. Actua com la funció lògica AND i gràficament es representa amb un símbol '→'.
- **Paral·lel**. Actua com una seqüència però executa tots els fills alhora. Retorna fallada si un dels seus fills retorna fallada.
- **Decorador**. Es un tipus de tasca amb un sol fill. El decorador n'altera el comportament. És útil per canviar o afegir funcionalitats al comportament per defecte dels nodes. Es poden programar decoradors amb funcions diferents segons el comportament que es vulgui canviar.
    - Interruptor. Controla la finalització de la tasca que en penja. Actua de forma transparent fent passar el retorn del valor del seu fill normalment. Però si es requereix interrompre la tasca filla, aquest la finalitza i en retorna el valor desitjat. Va lligat amb una tasca que realitzi la interrupció (*perform interruption*). Això permet trencar la jerarquia de l'arbre i interconnectar processos horitzontalment.
    - Inversor. És una tasca simple que inverteix el valor de retorn del seu fill.
    - Repeat. Repetició o bucle. Executa el seu fill múltiples cops. Es pot determinar el nombre d'iteracions que ha d'executar.

- Until fail/success. Repeteix l'execució del fill fins que aquest retorna fallada o èxit.

* Tasques sense fills. Són les fulles de l'arbre.
    - Condicions. Realitzen un test sobre alguna propietat del sistema i no alteren l'entorn. No tenen l'estat execució. Retornen immediatament èxit o fracàs. El símbol que s'usa és un oval.
    - Accions. Com indica el nom, fan algun canvi, alteren el sistema. Cada node corresponent a una acció és un requadre que conté un nom descriptiu.
    - Wait. Tasca que atura l'arbre un temps especificat i retorna èxit.
    - Perform interruption. Tasca que activa la interrupció definida per un decorador en un altre punt de l'arbre.

L'arbre, que té la tasca com a element principal, estalvia haver de definir cada estat i les seves transicions explícitament. Això simplifica el diagrama quan aquest comença a ser gran en comparació amb *statecharts*. Aquests últims acaben sent un garbuix d'estats amb totes les possibles transicions, en canvi, els arbres de comportament tenen una estructura més modular on les branques són independents i poden ser reutilitzades o modificades afectant en menor mesura la resta del comportament.

A continuació es mostra un exemple senzill d'un arbre de comportament en que un microcontrolador genèric realitza una lectura del nivell d'aigua en un dipòsit i engega i apaga la bomba per mantenir-ne el nivell.

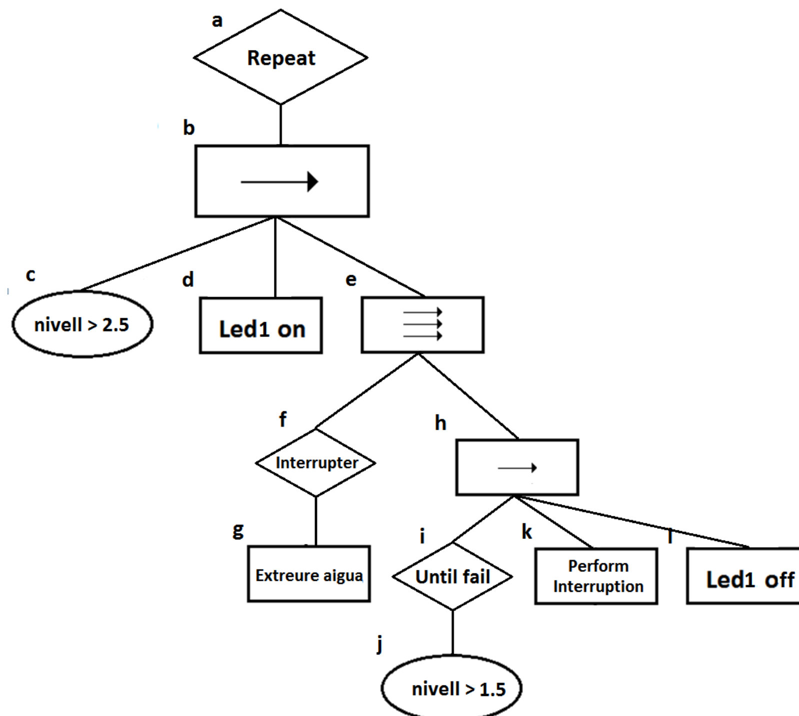



**Figura 1. Arbre de comportament per controlar el nivell d'un dipòsit.**

L'arbre s'executa de dalt a baix i d'esquerra a dreta. El node arrel és un decorador. Executa en bucle indefinit la branca que té a sota. El seu fill (*b*) és un selector que executarà cada un dels fills per ordre fins que hi hagi un retorn fals (failure). El node *c* llegeix el nivell de l'aigua; si la condició es compleix passa al següent node que és una tasca per engegar un led o llum pilot.

El node paral·lel (*e*) executa de forma concurrent tots els seus fills. Aquesta branca executa alhora la bomba extractora (*g*) i comprova que l'estat del nivell de l'aigua no baixi d'un mínim(*j*). Per poder fer això cal poder interrompre l'execució normal de l'arbre. A la tasca 'Extreure aigua' se li ha afegit un decorador (*f*) que pot interrompre l'execució del seu fill i retornar èxit. És important el valor del retorn de (*i*) perquè si s'interrompés l'execució de la tasca i aquesta retornés fals s'aturarien sense control tots els nodes *k* i *j,* i l'error es propagaria al node paral·lel (*e*) i tots els seus possibles fills. El fil que s'executa a partir d' *h* comprova constantment el nivell i quan baixa d'1.5 la tasca *k* crida la interrupció aturant la bomba (tasca *g*) i apaga el led indicador (*j*).

En un cas real aquest tipus de sistema s'hauria d'afegir accions alternatives en cas que alguna de les tasques retornés error. Part d'aquest control d'errors es pot fer programàticament en cada node mitjançant les interfícies i valors de retorn possibles que es poden donar, explicitar-ho en el mateix arbre de comportament o una combinació d'ambdues.

# 3. Especificació

Els elements clau que ha de complir aquest projecte s'especifiquen a continuació.

E1. Seleccionar una plataforma hardware on executar els programes i un conjunt de llibreries sobre les quals es treballarà.

E2. Dissenyar i implementar una sèrie de classes que representen un arbre de comportament i poden ser executades en el hardware escollit.

E3. Dissenyar un editor gràfic que implementarà els arbres de comportament com a eina per estructurar el programa. L'editor ha d'integrar funcionalitats necessàries per generar codi, com són:

> E3.1. Poder afegir el codi corresponent a la tasca que fa cada node per construir l'arbre que representa el comportament del programa.
>
> E3.1. Generar el codi del programa partir de l'arbre de comportament i el codi especificat per l'usuari.
>
> E3.1. Disposar d'eines de verificació i detecció d'errors de disseny i ús de nodes.
>
> E3.1. Incorporar a l'editor les fases de compilació per la plataforma hardware en que es treballa. Un cop dissenyat i generat el codi de programa si es vol que es pugui generar també els executables.

E4. Comprovar els resultats generant un conjunt de programes d'exemple executables en el hardware escollit. Es vol validar d'aquesta manera que l'entorn de desenvolupament és prou útil com a eina per dissenyar el comportament d'un microcontrolador a alt nivell i facilita la codificació del programa corresponent.

Al projecte s'hi poden afegir més llibreries en quant a les eines disponibles a nivell de la programació del dispositiu. Es pot donar suport a altres plataformes i els llenguatges que usen, o fins i tot usar altres estructures diferents als arbres de comportament per estendre les possibilitats del disseny.

En ser una part important del projecte, la interfície gràfica ha de ser robusta i fàcil d'usar per part de l'usuari.

# 4. Disseny

L'apartat de disseny de l'entorn de desenvolupament es divideix en les fases principals per les que passa l'editor des que l'usuari dibuixa el comportament de del seu programa fins que s'obté el resultat final, el codi de programa o executable.

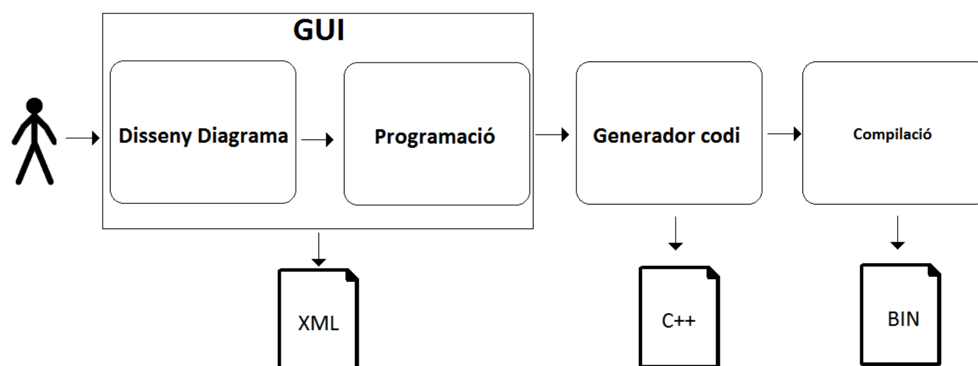



Figura 2. Principals mòduls i fitxers de sortida del programa.

## 4.1. Interfície gràfica

Les funcionalitats principals de l'editor s'ordenen a continuació.

El software a desenvolupar consta d'un espai de treball semblant a una eina de dibuix o disseny de diagrames constituït per un conjunt d'eines per dibuixar l'esquema de l'aplicació amb blocs que es poden moure, enllaçar i anomenar com mostra la figura 3 (b).

Aquest programa, com tot IDE, disposa de funcions per copiar o enganxar nodes o part de l'arbre i d'altres opcions com ara desfer els canvis anteriors o inserir comentaris.

Els blocs que formen el disseny de l'arbre es poden obrir o expandir per afegir el codi necessari per al programa en que s'està treballant mitjançant un editor de text.

El software analitza el disseny per fer un control d'errors ja que cada node té característiques diferents i s'ha de determinar si una construcció en concret és correcta o no. Per exemple, els decoradors només poden tenir un fill o altres nodes com les accions no poden tenir fills.

A partir de l'arbre de comportament i la programació de cada node, l'editor és capaç de generar el codi de programa equivalent al del disseny especificat.

Es dóna la possibilitat de desar el projecte obert i un cop finalitzat, desar el codi per exportar-lo a una altra eina o compilar-lo  per ser executat en el hardware desitjat. (Veure fitxers de sortida de la figura 2).

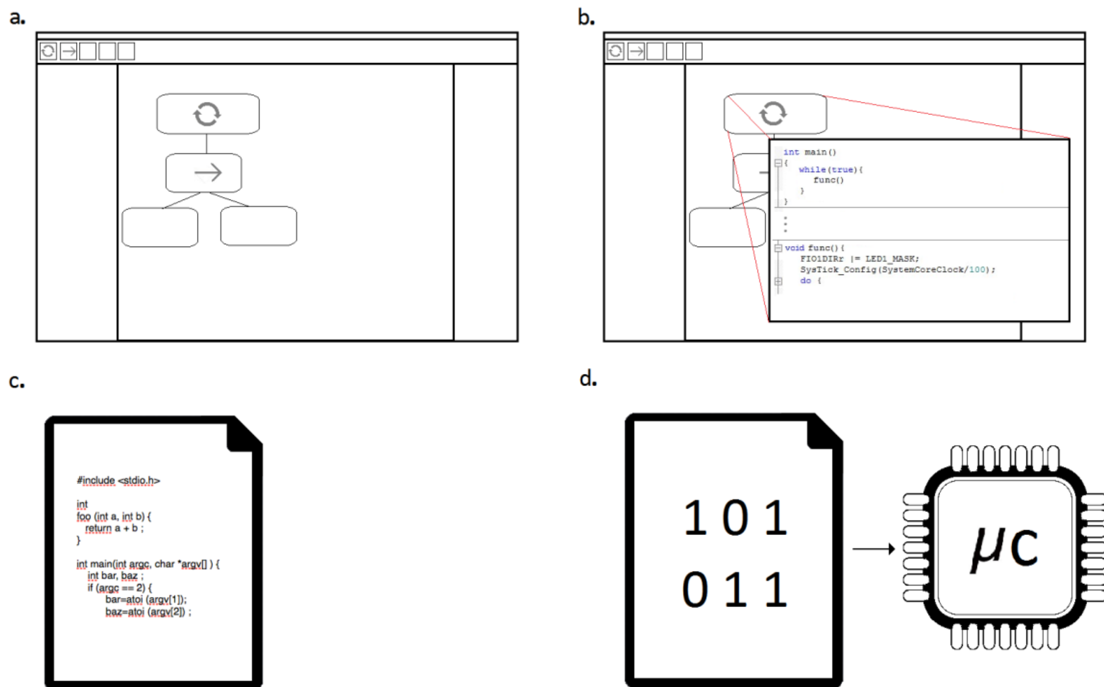



Figura 3. Exemple de la UI i funcions principals. (a)Disseny de l'arbre de comportament, (b) programació del node, (c) generació del codi font del programa, (d) binari o executable.

## 4.2. Programació

A mesura que es va dissenyant l'arbre de comportament es poden programar les tasques que fan els nodes.

Els nodes que són fulles de l'arbre són les tasques que s'han de programar. Els altres nodes són compostos, agrupen altres tasques o són decoradors i ja tenen una funcionalitat predefinida.

Tot i que no s'han de programar, alguns dels decoradors poden ser parametritzats. Per exemple s'ha d'especificar el nombre d'iteracions que una tasca de tipus *repeater* ha d'executar el seu fill.

Cada node programable correspon a una funció o mètode en el llenguatge de programació que s'utilitzi. Cada node disposa d'accés a variables globals si s'han especificat i obligatòriament un valor de retorn que usa l'arbre per determinar l'estat o resultat de l'execució de cada node i avançar en l'execució del programa.

En seleccionar un node s'obre una finestra nova amb un editor de text preconfigurat amb l'esquelet de la funció corresponent. Aquesta finestra es comporta com l'editor de text en un IDE qualsevol, oferint ressaltar text, copiar i enganxar, desfer, refer o desar.

La programació per part de l'usuari és senzilla. El procés per passar cada una de les funcions i nodes al codi final del programa és automàtic, transparent al programador. La metodologia a  usar s'introdueix en la secció següent.

## 4.3. Generador de codi

Els tipus de node que conformen l'arbre es poden diferenciar en dos principalment. Les tasques que són fulles i les tasques que no ho són. Les fulles de l'arbre s'han de programar, és a dir, se'ls ha de donar una funcionalitat. Els altres nodes són nodes compostos; decoradors, seqüències,  etc. Aquests nodes compostos agrupen altres tasques i ja tenen una funció predefinida.

Un paradigma de programació adequat per passar el disseny de l'arbre de comportament a codi és l'orientat a objectes.

Es pot entendre cada node de l'arbre com una instància d'una classe corresponent a una tasca genèrica. En el cas de les fulles de l'arbre, aquesta instància té definida una funció que el programador ha d'omplir, un retorn i uns mètodes auxiliars per interrompre o modificar-ne l'execució.

Segons el llenguatge de programació i les capacitats del compilador per la plataforma destí de l'aplicació hi ha diverses tècniques per implementar aquesta metodologia. Un exemple en són els mètodes virtuals a implementar en classes filles o  treballant a un nivell més baix punters a funcions.

D'aquesta manera s'obté l'estructura de l'arbre com un objecte (node arrel) que executa la instància d'un altra classe (primer fill) que alhora n'executarà d'altres fins arribar a les fulles de l'arbre. Quan les fulles acaben propaguen el resultat cap als seus pares de forma recursiva.

La sortida en aquest pas és el codi de programa complet generat i llest per ser executat en el hardware elegit. S'ha de poder exportar com a fitxer de text en el llenguatge emprat pel microcontrolador per ser modificat manualment en un editor de text si és necessari. En qualsevol cas s'ha de desar finalment en un fitxer al disc des de l'entorn de desenvolupament.

## 4.4. Compilació execució

L'última fase correspon a  la compilació del programa que s'ha generat amb l'entorn de desenvolupament.

Un cop s'ha obtingut el codi font del programa a partir de l'arbre dissenyat es pot recórrer a una eina externa, un compilador com els que es solen oferir amb els microcontroladors. Ara bé, l'objectiu del projecte és facilitar i automatitzar tot el procés de desenvolupament,  així que s'incorpora un compilador a l'entorn de desenvolupament per generar el binari pel microcontrolador. L'IDE pot disposar d'opcions per al compilador perquè l'usuari pugui configurar certs paràmetres.

L'últim pas que pot fer el programa, si es desitja, és copiar l'executable al microcontrolador si aquest es troba connectat a l'ordinador mitjançant una interfície com la d'un USB o un port sèrie.

# 5. Implementació

Com s'ha explicat en l'especificació del projecte, aquest s'ha d'aplicar a dues plataformes diferents. El codi s'ha d'executar sobre un microcontrolador però primer s'ha de dissenyar/implementar usant un entorn de desenvolupament en un ordinador de sobretaula. Per tant les eines i tecnologies (i llenguatges emprats en desenvolupar el projecte) són independents.

Al primer apartat d'aquesta secció s'explica com s'han implementat els arbres de comportament per ser executats en un microcontrolador. El segon apartat explica el funcionament de l'entorn gràfic. Seguint aquest ordre s'entendrà millor la interacció amb l'entorn de desenvolupament i com aquest manipula l'arbre de comportament i en genera el codi corresponent.

## 5.1. Implementació de l'arbre de comportament

La plataforma hardware escollida en que s'han d'executar els programes dissenyats usant arbres de comportament és el microcontrolador Mbed LPC1768 del fabricant NXP.

Les característiques principals d'aquest microcontrolador són les següents[ii]:

- CPU ARM Cortex-M3 96MHz 32 bit.

- 512KB Flash

- 32KB RAM

- Interfícies entrada/sortida: CAN, SPI, I2C, ADC, DAC, PWM, Ethernet, USB.

La connexió amb l'ordinador per transferir fitxers és a partir d'un connector USB que a part de l'accés a la memòria flash es pot usar de port sèrie virtual per enviar i rebre dades amb l'ordinador.

El llenguatge de programació per a la plataforma és C++ i es programa a través d'un IDE en línia. Amb el navegador d'internet s'hi pot accedir, crear projectes i compilar-los. El compilador és personalitzat i usa com a base armcc i incorpora l'estàndard C++03[iii].

La comunitat de desenvolupadors ha creat un compilador que incorpora les llibreries i eines per generar els binaris pel microcontrolador. El més important és que el codi sigui compatible amb l'eina oficial però s'ha intentat simplificar les tècniques i usar

l'estàndard més antic en la programació del projecte per que sigui funcional en les dues.

En tenir un hardware i llibreries diferents a les d'un PC també s'ha d'adaptar la programació a les capacitats de la plataforma, per exemple la impossibilitat de combinar interrupcions amb la programació paral·lela o el mateix maneig dels fils.

La memòria RAM disponible és la principal limitació a l'hora de programar per aquests dispositius. En usar llenguatges d'alt nivell, disposar de llibreries avançades per manejar dades i l'ús de programació orientada a objectes, es pot perdre la noció de la quantitat de memòria usada quan es comencen a instanciar objectes o col·leccions de dades.

Implementació dels nodes

S'ha considerat que la forma més natural de codificar els nodes de l'arbre és en classes. Cada tipus de node és una classe.

La classe base és Node. Aquesta classe ha d'implementar un sol mètode que tots els altres nodes sobreescriuran per que l'arbre funcioni i es propaguin els resultats. El node base defineix un únic mètode virtual *run* de retorn booleà. Aquest executarà cada tasca corresponent als diferents nodes.

```
class Node {
    public:
        virtual bool run() = 0;
};
```

**Figura 4. Node base.**

Nodes compostos

Els nodes compostos permeten agrupar tasques per ser executades en sèrie o paral·lel, ordenades o no i aturar-se segons el valor de retorn individual dels nodes fills. Poden emular les funcions lògiques AND i OR.

```
class CompositeNode : public Node {
    private:
        vector<Node*> children;
    public:
        const vector<Node*>& getChildren() const {
            return children;
        }
        void addChild (Node* child) {
            children.push_back(child);
        }
};
```

**Figura 5. Node compost.**

La implementació fa l'ús de la clase std::vector necessària per mantenir una llista de nodes, la possibilitat d'afegir-ne o obtenir la llista. És útil ja que té mida variable i internament està construït igualment amb un objecte de tipus array.

Les classes que defineixen un node compost i s'usen en l'arbre són el selector, la seqüència i el paral·lel. A continuació s'expliquen els nodes que s'executen seqüencialment, sense fer ús de fils.

```
class Selector : public CompositeNode {
    public:
        virtual bool run() {
            vector<Node*> v = getChildren();
            for (int i = 0; i < v.size(); ++i) {
                if(v[i]->run())
                return true;
            }
            return false;
        }
};
```

Figura 6. Node Sel·lector.

El node del codi anterior (figura 6) correspon a un sel·lector. Com que és un node que ha de formar part de l'arbre, ha d'implementar la funció run. Recorre el vector de nodes i els executa seqüencialment, retornant true si un d'ells te èxit.

El següent node correspon a la seqüència (figura 7). Com en el node sel·lector, recorre els fills i els executa seqüencialment fins que un retorna fals o acaba de recorre'ls tots amb èxit, retornant *true*.

```
class Sequence : public CompositeNode {
    public:
        virtual bool run() {
            vector<Node*> v = getChildren();
            for (int i = 0; i < v.size(); ++i) {
                if(! v[i]->run())
                return false;
            }
            return true;
        }
};
```

Figura 7. Node Seqüència.

Aquest tipus de node es pot usar amb una tasca condició al principi d'una seqüència per crear l'equivalent a una declaració de tipus *if*. Si s'encadenen seqüències sota un selector el comportament seria el d'una declaració *if*, *else if*.

Decoradors

El node decorador hereta de Node i modifica el comportament d'un únic fill. Per tant, aquesta classe té un fill i una funció per assignar-li aquest fill. Tots els decoradors en seran una subclasse i implementaran la funció run amb el comportament programat pel seu fill.

```
class DecoratorNode : public Node {
    private:
        Node* child;
    protected:
        Node* getChild() const {
            return child;
        }
    public:
        void setChild (Node* newChild) {
            child = newChild;
        }
};
```

Figura 8. Node Decorador.

S'han creat els decoradors necessaris per dur a terme les funcionalitats i estructures de comportament bàsiques que es poden realitzar en un programa.

El primer decorador és el node especial Root que fa d'arrel del programa. En el codi C++ que del microcontrolador només té la funció d'iniciar l'execució de l'arbre cridant el mètode *run* del seu fill. Correspon a la figura de sota (figura 9).

```
class Root : public DecoratorNode {
    public:
        virtual bool run()  {
              return getChild()->run();
        }
};
```

Figura 9. Node Root o arrel.

El decorador inverter nega el resultat de l'execució del seu fill.

```
class Inverter : public DecoratorNode {
    private:
        virtual bool run()  {
              return ! getChild()->run();
        }
};
```

Figura 10. Node Inverter.

Un altre dels decoradors és *succeeder* que retorna èxit (*true*) independentment del resultat de l'execució del seu fill.

```
class Succeeder : public DecoratorNode {
    private:
        virtual bool run()  {
              getChild()->run();
              return true;
        }
};
```

Figura 11. Node Succeeder.

Per crear bucles de l'estil *while* hi ha la classe corresponent al node *repeater*. El constructor d'aquesta classe admet el nombre de repeticions que es desitja o el valor -1 per indicar que el bucle és infinit.

```
class Repeater2 : public DecoratorNode {
    private:
        int numRepeats;
        static const int INFINITY = -1;
    public:
        Repeater2 (int num = INFINITY) : numRepeats(num) {}
        virtual bool run()  {
            if (numRepeats == INFINITY)
                 while (true) getChild()->run();
            else {
                for (int i = 0; i < numRepeats - 1; i++)
                    getChild()->run();
                return getChild()->run();
            }
        }
};
```

Figura 12. Node Repeater

Aquest decorador és molt usat com a fill immediat del node arrel quan el microcontrolador ha de fer una tasca que es repeteix indefinidament. El nom de la classe en la figura 12 es degut a que l'eina no oficial per compilar en l'ordinador (fora de línia) no admet la paraula clau Repeater com a nom de classe.

El node repeat until fail executa en bucle el seu fill i només s'atura quan el retorn és fals. Després retorna true.

```
class RepeatUntilFail : public DecoratorNode {
    public:
        virtual bool run() {
            while (getChild()->run()) {}
            return true;
        }
};
```

Figura 13. Node RepeatUntilFail

## Fulles

S'han considerat tres classes diferents com a fulles de l'arbre. La primera d'elles és el waiter.

```
class Waiter : public Node {
    private:
        int waitTime;
    public:
        Waiter (int waitTime) : waitTime(waitTime ) {}
        virtual bool run() {
            wait_ms(waitTime);
            return true;
        }
};
```



Figura 14. Node Waiter

La classe Waiter en la figura superior (figura 14) hereta de node directament. El constructor rep el temps d'espera en milisegons com un enter.

Waiter podria ser implementat com a decorador però s'ha considerat que en l'àmbit de disseny queda més clar l'efecte que produeix a les fulles germanes, les del seu mateix nivell. A més en quan a funcionalitat es pot triar en quina posició es vol executar l'espera quan es troba entre les fulles filles d'un node compost com el d'un selector.

Les dues últimes classes que representen fulles en l'arbre de comportament són acció i condició. La representació interna és la mateixa ja que duen a terme alguna tasca (acció) o executen un test (condició) i retornen un valor booleà segons l'èxit o el fracàs. L'única funció que Implementen és la funció *run*. Les diferències entre les dues classes es troben en el disseny gràfic de l'arbre més que en l'actual implementació C++ dels nodes. En el cas que es volgués fer més complexa la implementació de l'arbre afegint més valors possibles en l'estat d'execució dels nodes (*running*, *paused*, etc.), es podria considerar que la condició té un retorn immediat cert o fals però una acció pot passar per diversos estats.

```
class AccioN : public Node {
    public:
        virtual bool run() {
            led1 = 1;
            return true;
        }
};
```

**Figura 15. Exemple de node Acció.**

```
class CondicioN : public Node {
    public:
        virtual bool run() {
            return (led1 == 1);
        }
};
```

**Figura 16. Exemple de node Condició.**

La figura 15 mostra una acció, engega un led i retorna *true* suposant que hi ha èxit. La figura 16 correspon a una condició i retorna l'estat del led que pot ser engegat o apagat. Aquests dos últims codis són exemples de la possible implementació d'una tasca. Aquestes dues classes s'implementen de nou amb un nom diferent quan es crea una nova acció o condició. Com s'explicarà en l'apartat de l'entorn de disseny, les accions i condicions són un una plantilla que s'omple amb el codi de la funció *run* quan l'usuari crea i programa una nova tasca.

Fins ara els nodes implementats permeten implementar un arbre de comportament que s'executa seqüencialment. Tot i això, és possible programar fils i interrupcions de hardware en els nodes acció. També és pot fer de ús dels nodes selector i seqüència i de tots els decoradors que conjuntament poden representar les funcions d'un programa qualsevol, incloent construccions condicionals i bucles i funcions

A continuació s'explica com s'han creat els nodes necessaris per dissenyar comportaments que requereixin fer ús de tasques en paral·lel. S'han programat per separat ja que existeixen algunes diferències i limitacions a l'hora d'usar aquests nodes.

Nodes per execució en paral·lel

Com es comentava al principi d'aquest apartat, la implementació en un microcontrolador té certes restriccions. La limitació més important detectada en el disseny de les classes corresponents als tipus de nodes és la creació i maneig de fils. A continuació s'esmenten les característiques principals de les llibreries per crear fils en l'entorn *mbed* destinat al microcontrolador elegit.

1) La llibreria amb què es treballa és una implementació bàsica d'un RTOS o sistema operatiu en temps real per processadors Cortex ARM.

2) Aquesta implementació no permet crear ni instanciar fils des d'àmbits (*scope*) no estàtics.

3) Només es poden crear fils amb una crida a l'objecte Thread, el constructor del qual admet un punter a una funció sense retorn (*void*) i un punter als paràmetres. També opcionalment es pot especificar la prioritat d'execució o la mida de la pila a usar.

La primera característica fa que degut a la implementació del sistema operatiu en temps real en aquest microcontrolador, les interrupcions per hardware com les que es produeixen per flanc en una entrada digital no funcionin. Accions com la detecció de flancs s'han de fer per *polling*. A més el maneig de fils és més limitat, tant per les opcions que ofereix el software com el hardware.

La segona i tercera impedeixen que la instància d'una classe corresponent a un node pugui iniciar-se en un nou fil com es podria fer per exemple en java extenent runable i implementant una funció *run*.

Poder executar una funció d'una classe en un nou fil faria que el codi i la crida del mètode *run* d'un node executat en sèrie fos la mateixa que en paral·lel.

Això tampoc permet, a priori, vincular un fil a un objecte com el que representaria un node.

Per poder executar nodes en paral·lel s'han hagut d'implementar els nodes paral·lel i fer grans modificacions a les classes acció i condició.

El resultat final és funcional però són classes complexes i tenen un impacte sobre la memòria molt més gran. Les accions i condicions són també transparents al dissenyador/programador i poden ser usades tant en execució normal com en paral·lel.

En les dues figures següents es mostren les definicions de la classe paral·lel i acció, s'explica com han estat implementades i com funcionen. (El codi complet es troba als annexos 1 i 2 donada la seva extensió).

```
class Parallel : public CompositeNode {
    private:
        Queue<bool, 16> resQueue;
        Mutex res_mutex;
        int results;
        int consumed;
    public:
        Parallel();
        virtual bool run();
        void setResult(bool* b);
        virtual bool terminate();
};
```

Figura 17. Node Paral·lel

```
class Action : public Node {
    private:
        static void threadStarter(void const *p);
        void runable();
        Thread *_thread;
        bool _isThread;
        Parallel *_parent;
    public:
        Action(Parallel *parent=NULL);
        virtual bool run();
        bool returnValue;
        virtual bool terminate();
};
```

Figura 18. Node Acció per ser executada tant en paral·lel com seqüencialment.

El primer codi (figura 17) és el node compost paral·lel. Implementa la funció *run* com tots els altres nodes. Aquesta funció, com en el node seqüència, inicia tots els fills però ho fa en fils nous.

La diferència és que els fils nous no poden retornar cap valor i se'n perd el control ja que sinó s'haurien de mantenir referències a tots els fils i crear mètodes per treballar-hi. I a més, en un moment determinat, només s'executa un fil en el processador.

El comportament del paral·lel es du a terme des de la funció *run*.

El que s'ha decidit fer és iniciar tots els fils, amb una referència del pare (node paral·lel) i posar a dormir el node paral·lel. Aquest s'espera en un bucle que llegeix d'una cua que ofereix el sistema operatiu per comunicació entre fils. Cada cop que un fill acaba notifica al pare i aquest llegeix el resultat.

Aquest node actua com s'ha especificat que es comporta el node paral·lel; si un dels fills acaba en fracàs s'interromp la resta; sinó, va esperant fins que acaben tots. Quan tots els fils han acabat retorna cert actuant de forma transparent a la resta dels nodes, tenint en compte, que la implementació és diferent a la resta de nodes de l'arbre.

Implementa un *mutex* per protegir la cua de resultats, una funció *setResult* per tal que els fills actualitzin la cua. També afegeix una nova funció, necessària en l'execució de fils que s'anomena *terminate*. Serveix per finalitzar un node si es necessari per mantenir el control de l'execució de l'arbre, per exemple quan un dels fills ha retornat fals i ja no cal seguir executant els altres nodes.

La classe Acció (figura 18) ha estat modificada perquè pugui funcionar com a filla en un node paral·lel o normal, de forma seqüencial com fins ara. Degut a les limitacions a l'hora de crear fils en objectes s'han analitzat a fons les possibilitats del llenguatge i la API per aconseguir emular el funcionament desitjat.

El constructor de la classe té un sol paràmetre que és un punter al node pare. Si és una execució en paral·lel s'explicitarà aquest valor i quedarà reflectit en la variable booleana *_isThread*. En el mateix constructor s'inicia el fil si és necessari.

Si s'ha de crear aquest nou fil, s'inicia amb la funció *threadStarter* que és de la mateixa classe Acció però definida com estàtica, la referència al nou fil es desa a **_thread*. El paràmetre que se li envia a la funció és la instància de l'Acció que l'està iniciant usant la paraula *this* (figura 19).

```
if (_isThread)
      _thread = new Thread(&Action::threadStarter, this,
osPriorityNormal,512);
```

**Figura 19. Inicialització del fil si l'acció ha de ser executada en paral·lel.**

Un cop inicialitzat el fil en una funció estàtica, aquest, mitjançant la referència a l'objecte Acció crida la funció *runable* que realitzarà la tasca definida en el node. El fil entra en espera a l'inici de la funció *runable*.

Quan el fil principal, des del node paral·lel, es crida la funció *run* de l'acció, aquest, envia un senyal per despertar el fil que seguirà executant *runable* (figura 20). La funció s'executarà normalment en un fil i quan acabi escriurà el valor de retorn al node paral·lel amb la funció *setResult* explicada anteriorment.

```
if(_isThread)
      _thread->signal_set(START_THREAD);
else
      runable();
```



**Figura 20. Si s'ha d'executar en paral·lel es fa un *signal* al fil que executa *runnable* igualment.**

Aquesta implementació permet usar la mateixa funció (*runable*) en una execució concurrent o no. Tot i això el codi necessari fa més difícil d'entendre el funcionament de l'arbre a nivell de programació.

El consum de recursos és més elevat. Quant a la CPU no suposa gaire problema ja que és molt ràpida respecte al tipus de hardware del que solen disposar aquests dispositius. En canvi, per a la memòria RAM, sí que suposa una limitació important. En algun exemple que s'ha provat amb pocs nodes (6 nodes), l'augment en l'ús de la memòria ha estat del 20% al 40% en canviar solament un node seqüència per un de paral·lel per executar-ne les tasques filles concurrentment.

## 5.2. Implementació de l'entorn de desenvolupament

La segona eina implementada és l'editor per fer el disseny dels programes. Construeix el codi de programa a partir del dibuix d'un diagrama que representa un arbre de comportament. Genera el codi C++ amb la implementació de l'arbre de comportament programat fins ara.

Per crear aquest programa s'ha elegit el llenguatge de programació java ja que disposa de molta documentació i llibreries de tercers que en faciliten el desenvolupament. S'ha fet ús a més, de la llibreria *swing* de java en totes les interfícies gràfiques.

En els següents punts s'expliquen les característiques principals, com han estat programades les funcionalitats més importants del programa, i les eines usades.

### 5.2.1. Finestra principal. Àrea de disseny.

El programa té una finestra principal construïda en diverses capes. És la primera que es mostra en obrir el programa. Els elements de la interfície gràfica són objectes de la llibreria de java Swing i/o Awt i en molts casos s'han creat classes que hereten d'aquestes per modificar-les i donar-los les funcions desitjades.

A la part superior, s'hi ha situat el menú amb totes les accions per interactuar amb el programa.

A la part inferior s'ha creat un panell senzill per afegir-hi missatges i notificacions que el programa vulgui mostrar a l'usuari.

La zona esquerra de la finestra és un conjunt de pestanyes que contenen panells personalitzats per contenir i mostrar els diversos tipus de nodes que es poden usaren

el disseny de l'arbre de comportament. En iniciar el programa es llegeixen d'una llista els diversos nodes que s'han de carregar. La classe *LabelNode*, que representa cada node, manté un identificador pel tipus de node, la imatge i el nom del node. Aquests nodes s'afegeixen a les diferents pestanyes.

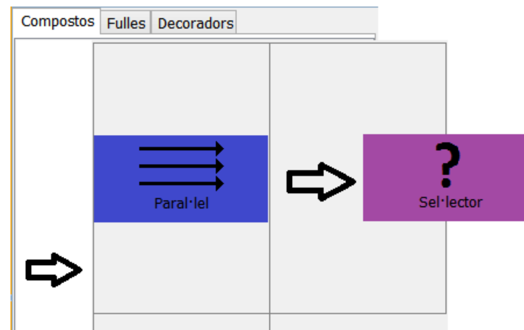



Figura 22. Capes de la pestanya nodes de la GUI

Els nodes de les pestanyes implementen la classe *Transferable* de Awt. Aquesta permet definir tipus de dades que es poden arrossegar i deixar anar per copiar-les o transferir-les (*Drag & Drop*). En aquest cas es transfereix l'identificador de tipus de node, un enter, que s'usa en una classe *Builder* per crear i afegir un node a l'arbre de comportament.

L'àrea que accepta els nodes arrossegats és la de la dreta de les pestanyes, que gairebé ocupa tot l'espai de la finestra i és la zona de treball pròpiament dita, on es pot dissenyar l'arbre.

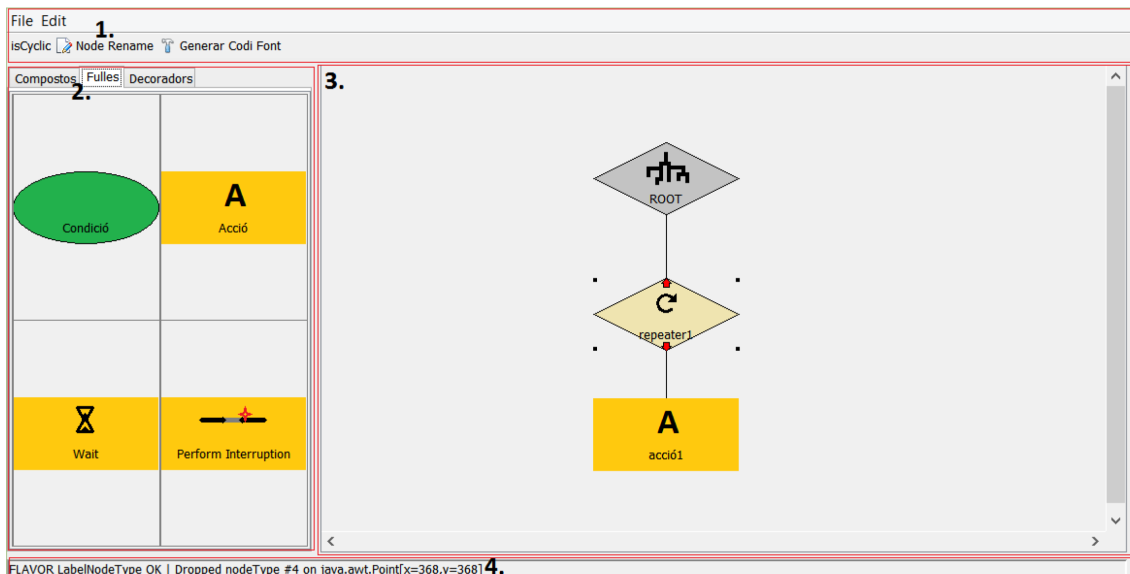



Figura 23 GUI. (1)Menú principal, (2)Pestanyes amb els nodes, (3) panell de treball, (4)Àrea de notificació.

L'àrea de treball ((3) en la figura anterior) és la classe més complexa. Estén un *Jpanel* amb *scroll* adaptatiu a la mida de l'arbre. Té tots els *listeners* necessaris per manipular els nodes: Seleccionar, arrossegar i deixar anar, doble clic per editar i dreceres de

teclat. També s'hi troben definides les funcions útils per afegir, eliminar, editar i modificar qualsevol node o enllaç de l'arbre. Conté una classe interna per canviar el nom dels nodes i s'han sobrecarregat les funcions de pintat per poder dibuixar els nodes i tots els elements gràfics que s'hi mostren.

L'arbre que s'hi dibuixa té dues vessants. L'apartat gràfic (interfície d'usuari) i la de l'estructura de dades i l'arbre en sí.

La classe *Node* és en la seva base semblant a la de *LabelNode*. Els nodes que es mostren a les pestanyes de l'esquerra.

S'ha fet ús d'herència en els objectes de java per crear els diferents tipus de nodes i característiques específiques a partir de l'objecte *Node* bàsic. La classe *Builder* crea segons l'identificador de tipus de node les classes filles corresponents. Aquest identificador és el que es passa quan s'usa arrossegar i deixar anar des de les pestanyes de nodes.

Es sobrecarreguen les funcions de pintar de tal manera que si s'ha seleccionat un node apareix el marc de la selecció com en el node "repeater1" de la figura 23. Si els nodes estan seleccionats també mostren dues fletxes vermelles que indiquen com es poden formar els enllaços amb altres nodes. La posició del ratolí en fer clic i arrossegar són variables de l'entorn que permeten saber si la s'ha d'arrossegar el node o crear l'enllaç amb un altre modificant el que es pinta automàticament.

Amb l'herència s'aconsegueix que cada tipus de node tingui interaccions diferents a la interfície gràfica. El doble clic, suprimir, o canviar el nom tindrà efectes diferents en alguns nodes. Per exemple el doble clic servirà per editar-ne el codi font o canviar alguna variable que n'afecta el comportament. En d'altres nodes no passarà res ja que no són modificables.

En el panell apareix per defecte el node arrel (ROOT) de l'arbre que és un node especial obligatori que no es pot esborrar. Els altres nodes es poden esborrar o reanomenar.

<u>Estructura de dades de l'arbre</u>

Quant a l'estructura de dades l'arbre es defineix a partir dels nodes i els enllaços entre aquests nodes. Cada node desa una llista de referències als seus fills i el seu pare.

Quan es creen els nodes s'afegeixen a un objecte Set on es desen i s'identifiquen amb una funció hash. D'aquesta manera es manté l'arbre com una sèrie de nodes i cada node té les referències als seus antecessors i predecessors. A més, per tal de ser

considerat un arbre de comportament ha de presentar certes restriccions. El graf ha de ser dirigit i no pot contenir cicles.

Per verificar que l’arbre està ben construït s'ha adaptat l'algorisme DFS (figura 3) que detecta cicles cada cop que s'intenta crear un enllaç nou a l'arbre. També es verifica que la direcció dels enllaços sigui correcta, és a dir que no s'enllacen fills entre ells i només tenen un pare.

Es disposa d'una arrel (node Root) com a base per l'arbre.

```
bool arbreBenFormat():
      Set visitats
      Lst roots    //Nodes sense pare

      for node in roots:
            if esCiclicAux (node):
                  return false
      return true


bool esCiclicAux(node):
      visitats <- node
      for fill in node:
            if fill is root: return true
            if not visited(fill):
                  return esCiclicAux (fill)
      return false
```

**Figura 24. *arbreBenFormat* comença recorrent tots els nodes lliures (sense pare) i fa crides recursives a la funció auxiliar comprovant si els fills ja han estat visitats**

Els enllaços que es desen en el node i que són les referències a altres nodes són els usats per pintar els enllaços en la interfície gràfica.

Segons el tipus de node que siguin tindran variables i funcions pròpies. Això també afecta a la interacció de l’usuari amb la interfície gràfica en que els nodes diferents tipus de nodes també tenen comportaments diferents.

### 5.2.2. Programació de l'arbre de comportament

El disseny del diagrama de l'arbre de comportament és la primera fase del procés. L'entorn de desenvolupament ajuda a programar en el llenguatge que usa el microcontrolador (C++) i personalitzar les tasques que fa cadascun dels nodes.

Alguns nodes no es poden modificar. Els decoradors *repeat until fail* o l'*inverter* no són modificables per que tenen una tasca predefinida que no cal parametritzar, un bucle fins fallar i invertir el valor de retorn respectivament. Els nodes compostos tampoc es

poden personalitzar directament, només mantindran un vector de nodes que executaran.

Altres nodes decoradors i totes les condicions i accions s'han programat per que s'hagin de modificar i codificar per donar-los un ús concret.

Com ha estat comentat en el punt anterior, les subclasses de cada node tenen paràmetres i funcions específiques. Quan es duu a terme l'acció d'editar, es crida el mètode corresponent del node seleccionat i s'activa l'edició en funció del tipus de node.

L'edició dels nodes obre una finestra nova amb un editor diferent segons el tipus de node. Aquests són els editors que s'han implementat per modificar i programar els nodes.

1. Editor de text.

Pels nodes més complexos s'ha creat un editor de text específic per a l'edició de codi. Sobre una finestra nova s'hi ha creat un menú i una àrea de text.

El menú conté els botons més bàsics d'un editor de text; refer, desfer i desar.

Per crear l'àrea de text s'ha usat una llibreria programada sobre l'element *TextArea* de swing que es diu *rsyntaxtextarea*[iv]. A part de *rsyntaxtextarea*, l'àrea de text no depèn de cap altre recurs extern.

Aquesta llibreria permet afegir un quadre de text com el *TextArea* i afegeix moltes altres funcions. S'han afegit uns *listeners* de teclat per fer ús de l'opció de desar una pila d'esdeveniments per fer i desfer el que s'escriu. També s'han activat les funcions d'editor de text enriquit. Com els editors de text Netbeans o Eclipse, l'editor de text pot ressaltar les paraules reservades dels diferents llenguatges de programació. En aquest cas s'ha activat el C++.  Ofereix també sagnia automàtica i plegar o desplegar fragments de codi.

L'editor de text carrega el codi font del node que s'està modificant. Inicialment es carrega el codi font d'una plantilla predefinida pel tipus de node. A partir d'aquest esbós es pot programar el codi necessari i desar els canvis que queden automàticament reflectits en l'estat intern del node actiu.

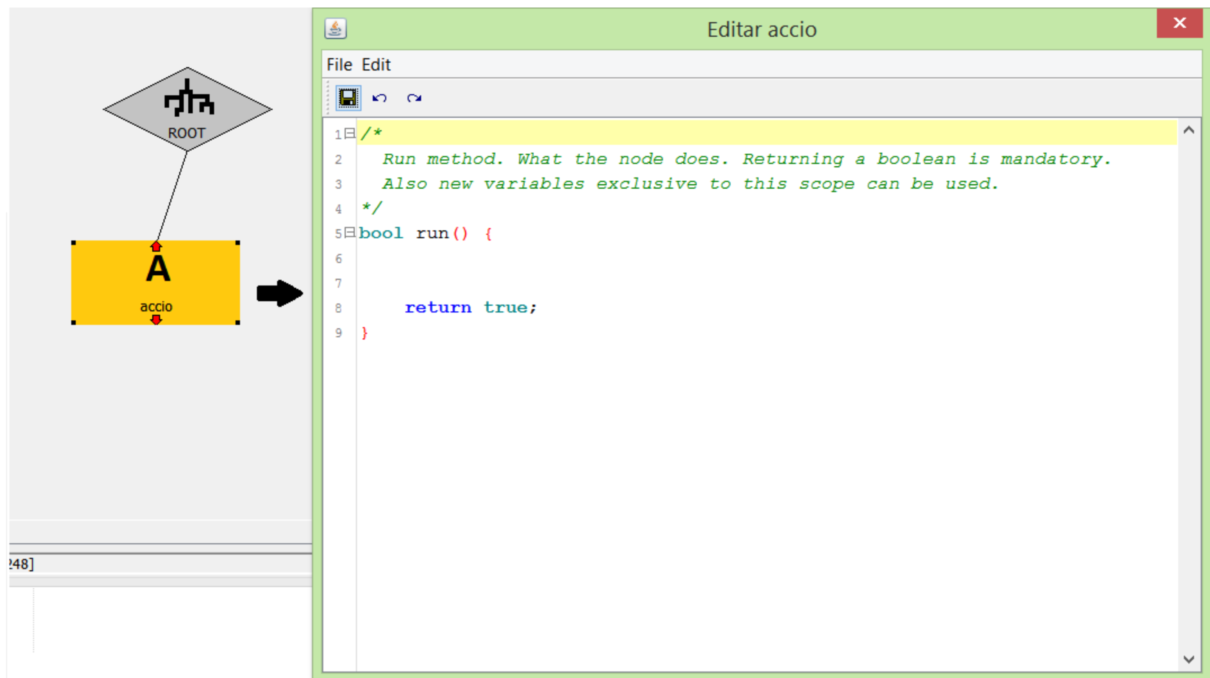



Figura 25 Editor de text d'un node de tipus acció.

Els nodes de tipus Root, Acció i Condició fan ús d'aquest editor.

El node arrel o Root equival a la capçalera del programa on es troben les llibreries usades, la funció main i les variables globals necessàries.

Acció i Condició són tasques que requereixen de programar la funció run del node que ha de retornar un valor booleà.

## 2. Editor d'espera

En el cas del node Wait s'ha creat una classe que obre una finestra semblant a la de l'editor de text. Podria servir per altres nodes que requerissin un paràmetre com un nombre.

En aquest cas té un camp on introduir el temps en milisegons que el programa haurà d'esperar. Fa les comprovacions bàsiques per verificar que s'ha introduït un valor correcte,que ha de ser positiu i trobar-se limitat a un enter amb signe de 32 bits.

Cal tenir en compte que els tipus de variable que usa la plataforma, com la mida en bits, on s'executarà el codi pot tenir unes limitacions diferents a les de l'entorn de disseny. Els tipus de dades usat s'adapta a les que el microcontrolador elegit requereix en la seva funció de *wait*.

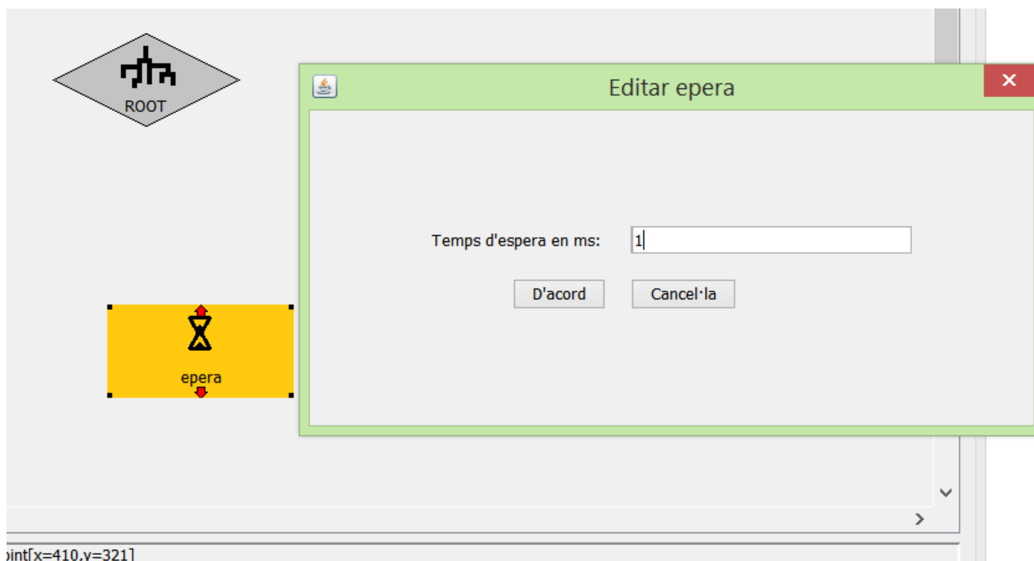



Figura 26. Captura de l'editor per inserir els valors del temps d'espera en ms.

3. Editor de nombre de repeticions

Aquest editor s'aplica al node Repeater que serveix per executar el node fill múltiples vegades, emulant un bucle.

Com l'editor de temps d'espera, té un camp de text on, en aquest cas, introduir el nombre de repeticions. També fa les verificacions necessàries al valor introduït que ha de ser enter entre -1 i 2^31 -1.

Molts programes en microcontroladors s'executen en un bucle infinit realitzant una tasca permanentment. Per aquest motiu s'ha inclòs un *checkbox* que indica que el bucle és infinit. En aquest cas desa el valor com -1.

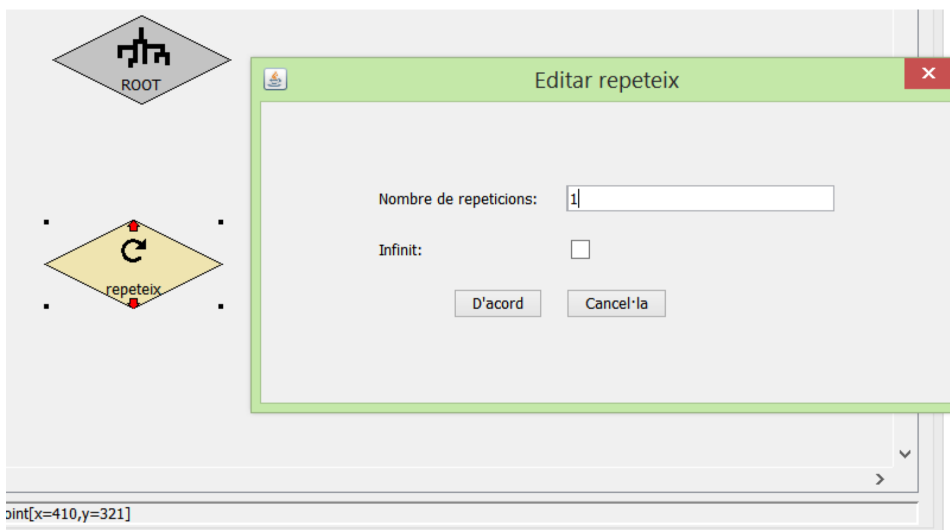



Figura 27. Editor per configurar les repeticions del node Repeater.

### 5.2.3. Generador del codi font a partir de l'arbre de comportament

La següent funció de l'entorn de desenvolupament després del disseny i programació de l'arbre i els nodes és obtenir el codi font corresponent per ser finalment exportat o compilat i executat en la plataforma hardware escollida.

S'ha creat una classe anomenada CodeGenerator que és instanciada quan es selecciona el botó generar codi font que hi ha al menú sobre l'àrea de disseny de la finestra principal.

Aquesta classe rep com a paràmetre el set de nodes que representa l'arbre de comportament carregat. Començant pel node arrel es va recorrent l'arbre de dalt a baix obtenint les dades de cada node i construint el codi font. A continuació s'explica el procés.

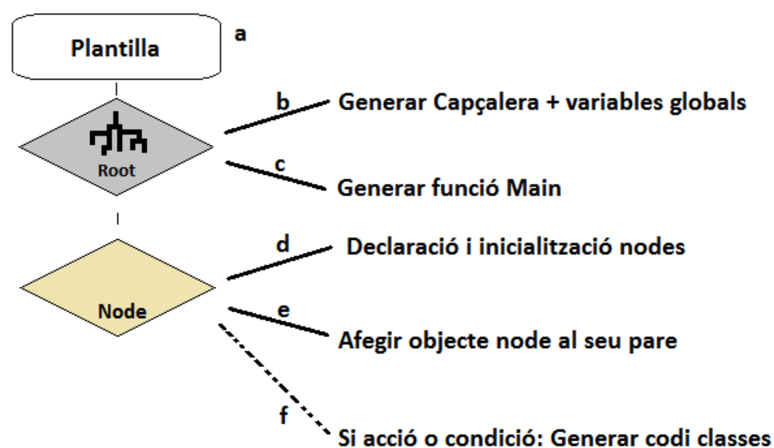



Figura 28. Procés de generació del codi font.

Es treballa a partir d'una plantilla que conté el codi de l'estructura del programa i uns identificadors que indiquen on inserir el codi font que genera el programa. A la plantilla hi ha tot el codi que no es variable: els *includes* imprescindibles i les classes que representen els nodes de l'arbre que no ha d'implementar l'usuari (figura 27 (a)).

A continuació es recorre l'arbre començant per l'arrel. El codi font d'aquest node ha d'haver estat implementat per l'usuari en l'etapa de programació. Al node Root s'hi poden definir les variables globals i altres capçaleres que es requereixin. Al node Root també s'hi defineix la funció Main del programa (figura 27 (b) i (c)). Qualsevol element que es vulgui programar separadament de l'arbre o variable que es desitgi manejar *manualment* es pot programar en aquest node.

S'analitza el text de la capçalera i la funció Main per separat i s'afegeix aquesta secció del codi a la plantilla.

A partir d'aquí com indica el punt d de la figura 27 per cada nou node de l'arbre es genera un String que representa la declaració i inicialització de la seva variable en C++ (figura 28).

```
ClassName* nodeName = new ClassName(params);
```

Figura 29

El tipus de classe la determina el tipus de node. El nom de la variable el determina el nom del node que primerament s'ha de netejar per evitar caràcters no vàlids (tractament semblant a un *sanitize* per entrar valors en una base de dades). Els paràmetres també els determina el tipus de node. En el cas dels decoradors que requereixen especificar algun paràmetre (temps d'espera en el node Wait), aquest paràmetre l'ha d'haver inserit l'usuari amb l'editor, s'obté del node i s'afegeix al constructor.

Les declaracions s'afegeixen a un String per ser incorporades juntes després a la plantilla.

El procés segueix afegint cada node com a fill del seu pare (figura 27 (e)) i per cada parella pare - fill es genera el codi corresponent en C++ (figura 29). Cal recordar que les classes ja tenen funcions de tipus *set* per afegir fills.

```
nodeName->addChild(childNode);
```

Figura 30

Les insercions també es desen seguides en un String per ser afegides a la plantilla a continuació de les declaracions dels nodes.

Com que l'arbre es recorre en ordre, les diferents declaracions i insercions també es fan en ordre i s'executen tal i com ha estat dissenyat el programa en el diagrama.

L'últim punt de la figura 27 (f) es realitza si els nodes són de tipus Condició i Acció.

Aquests nodes es representen amb una classe nova per cada node del disseny. L'usuari ha d'haver implementat la funció *run* i s'ha d'afegir la classe resultant al codi de programa. En aquest cas es parteix d'una plantilla per la classe Acció o Condició. Aquesta plantilla té uns identificadors que han de ser substituïts pel noms de la classe i el codi de la funció *run*.

Quan s'ha acabat de llegir l'arbre s'afegeixen a la plantilla totes les seccions del programa que s'han anat generant; la capçalera i variables globals, classes noves i la funció *main* a la qual s'hi afegeixen alhora les declaracions dels nodes, insercions als nodes i finalment la crida per iniciar l'execució de l'arbre: *root->run();*.

```
class ###CLASS_NAME### : public
Node {
    public:
        virtual bool run() {
            ###RUN_CODE###
        }
};
```

Figura 31. Exemple de la plantilla d'una classe nova.

Quan tot el codi ha estat generat, es retorna un String. L'entorn de desenvolupament obre una nova finestra amb l'editor de text que s'utilitza per programar els nodes.

Aquest editor és una modificació de l'anterior. Carrega el codi generat i se li han afegit algunes funcions noves. A més de modificar el codi generat a criteri del programador, es pot desar el codi en un fitxer de text o compilar-lo per generar l'executable pel microcontrolador tal com s'explica en la següent secció.

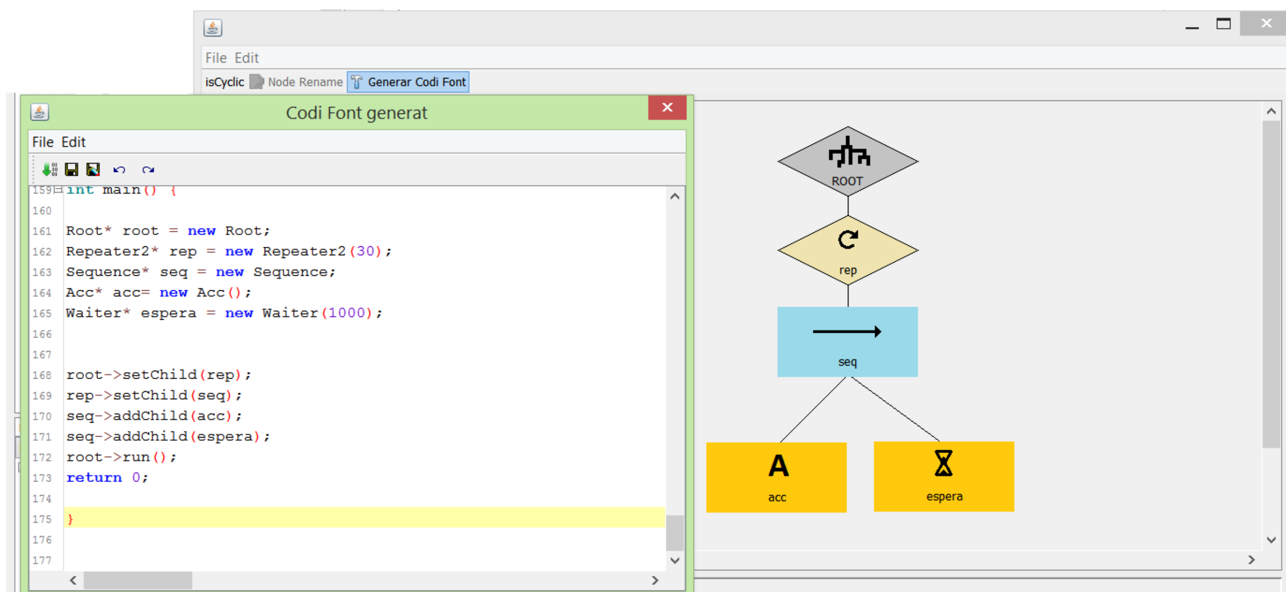



Figura 32. Finestra amb l'editor que mostra el codi font generat i les opcions per desar-lo o compilar-lo.

### 5.2.4. Compilació del codi font

Amb el codi construït queda l'opció de compilació per acabar el programa. Tot i que es pot desar el codi font en un fitxer de text i usar el compilador en línia que ofereix Mbed, és interessant poder-ho fer des del programa com fan altres entorns de desenvolupament integrats, sense dependre de connexió a Internet o eines externes.

Seguint la secció anterior, el codi font generat es mostra en un editor de text que té, a més de la funció de desar, un botó per compilar. Aquesta opció demana un lloc de destí pel fitxer compilat.

Per compilar s'ha creat una sèrie de classes que fan tota la feina automàticament. En aquesta fase del programa s'ha hagut de treballar a nivell de crides al sistema operatiu i mitjançant la consola. El compilador creuat que s'ha usat no és independent de la plataforma. Per aquests dos motius aquesta funció només és disponible en Windows.

El compilador creuat que ha estat desenvolupat per un membre de la comunitat de *Mbed* és *gcc4mbed*[iv].

Per començar el procés de compilació es desa el codi font en un fitxer temporal dins el directori del compilador.

Mitjançant la classe *Process* de java es crida la consola de Windows (cmd). S'ha creat també una classe auxiliar que s'anomena CommandTools que s'executa en paral·lel i obre un *stream* d'entrada i sortida per comunicar-se amb el procés de la consola, enviar instruccions i rebre'n el resultat.

A través de la consola s'inicien les variables d'entorn del compilador i es compila el codi font que si tot va bé genera un executable temporal en un directori dins del directori del compilador. Si hi ha algun error s'obté el codi d'aquest error amb la classe auxiliar i es notifica. Finalment es copia el fitxer executable al directori destí especificat i es netegen els fitxers intermedis.

Durant tot el procés de compilació una finestra emergent serveix de Log on el programa va abocant tots els procediments incloent els errors del compilador si sorgeix algun problema.

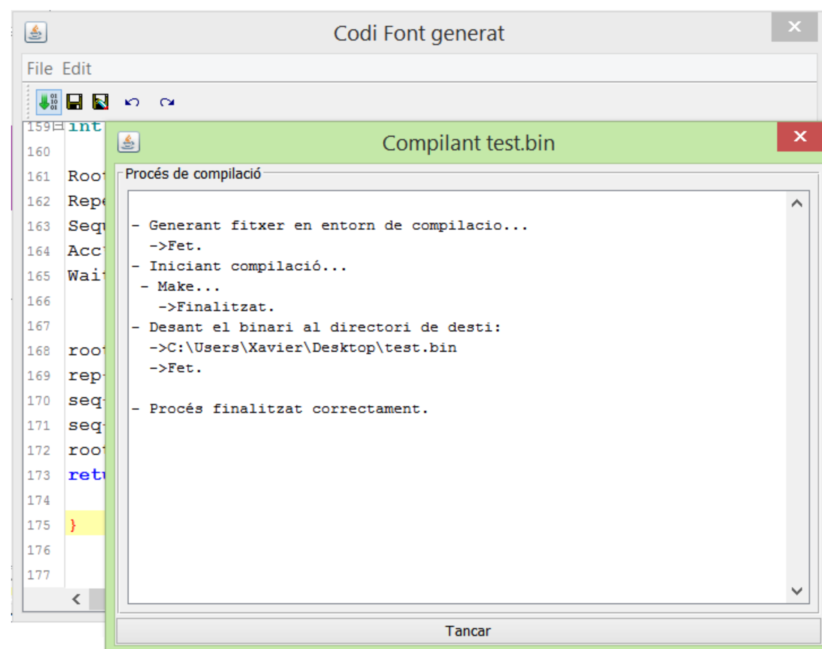



Figura 33. Finestra del procés de compilació del projecte.

### 5.2.5. Exportar el projecte

És important poder desar i exportar les dades d'un projecte i poder-les recuperar després. L'última de les funcions principals de l'entorn de desenvolupament és la funció per exportar i importar les dades d'un arbre de comportament.

El format que s'usa per desar l'arbre és un xml d'estructura molt senzilla. Les eines usades són les que proporciona javax.xml.

Per fer aquesta tasca s'han creat dues classes, Exporter i Importer que són dues classes estàtiques.

Exemple d'un sol node d'un arbre de comportament exportat:

```
<BTree>
  <node>
    <id>2</id>
    <nodetype>8</nodetype>
    <nom>rep</nom>
    <parent>1</parent>
    <children numchildren="1">
      <child>3</child>
    </children>
    <location>
      <x>291</x>
      <y>122</y>
    </location>
    <data><![CDATA[ ]]></data>
    <numrepeticions>30</numrepeticions>
  </node>
</BTree>
```

Figura 34. Exemple d'xml que representa un arbre format per un node.

A l'apèndix 3 s'hi troba l'xsd[vi] (XML Schema) que representa l'esquema de l'xml usat per desar els arbres de comportament en aquest programa.

<u>1. Exporter</u>

La classe Exporter ofereix una funció per desar l'arbre en format xml a partir del set de nodes i un fitxer destí.

Extreu els nodes del set un per un sense importar l'ordre, ja que la relació entre nodes va ja ve determinada pels seus enllaços. L'estructura de les dades de l'arbre és definida per l'etiqueta <BTree> i cada node per l'etiqueta <node>.

Per cada node es desa la seva informació genèrica; un identificador únic, el tipus de node, el nom, l'identificador únic del node pare i fills, la seva posició en l'àrea de treball en format de punt (x,y) i una etiqueta data que conté el codi font que s'hagi pogut programar en el node.

Finalment, si el node té més informació  es desen aquestes dades extra. Per exemple, en el cas del node Wait es desa l'enter que indica el temps d'espera.

Quan s'ha construït l'arbre, s'usa la classe *Transformer* que ofereix la llibreria per generar i formatar l'xml que es desa en el fitxer destí.

2. Importer

La classe Importer també disposa d'una sola funció. Aquesta requereix un fitxer origen i retorna una llista de nodes.

Retorna una llista amb tots els tipus de nodes i enllaços ja creats per ser afegits a l'àrea de treball de l'entorn de desenvolupament. És una mica més complexa ja que s'ha d'anar analitzant l'xml a mesura que es llegeix per crear els tipus de nodes i dades necessàries.

S'obre el fitxer d'origen i es van extraient els nodes amb les funcions que ofereix javax.xml.parser. De cada etiqueta que delimita un node se n'extreu l'identificador, el tipus de node, el nom, l'identificador del pare, la seva localització i el codi font.

En aquest punt s'usa la classe NodeSimpleFactory per crear un nou node del tipus necessari al que se li afegeixen totes les variables extretes de l'xml.

Si el node que s'ha creat es d'un tipus que ha de contenir informació extra com en el cas del node Repeater, que necessita el nombre de repeticions, es llegeix l'etiqueta corresponent en l'xml i s'afegeix a l'objecte.

Els enllaços entre nodes en el programa són referències entre els objectes Node però en l'xml es desen els identificadors numèrics d'aquests**.** Per tant per poder recrear aquests enllaços, quan s'analitza el fitxer, es desen parelles d'identificadors pare - fill per una banda. Per l'altra banda es manté un diccionari amb l'identificador numèric que representa el node(pare) com a clau i amb l'objecte Node recent creat com a valor.

Quan s'han acabat de llegir tots els nodes es recorren tots els enllaços i en el diccionari s'extreuen les referències als objectes que s'usen per enllaçar-los altre cop entre sí.

```
//Tree conté <id(int),Node>
for link in Links:
      Node p = tree.get(link.pare);
      Node f = tree.get(link.fill);
      p.setFill(f);
      f.setParent(p);
```

Figura 35. Recreació dels enllaços (referències) a partir dels identificadors llegits de l'xml.

La figura 34 mostra més clarament com es recreen els enllaços.  Quan s'ha acabat es desen els Nodes de l'arbre a la llista que la funció retorna.

El programa quan obté la llista de nodes neteja l'àrea de treball i la reomple amb els nodes nous repintant la interfície gràfica i netejant les variables de l'editor.

# 6. Test, proves, resultats i valoracions

El projecte s'ha desenvolupat dividint-lo primerament en dues parts ben diferenciades: la part corresponent a l'arbre de comportament pel microcontrolador i la de l'entorn per dissenyar els programes amb ordinador. Ambdues han estat dividides en mòduls sobre els quals s'ha anat iterant. Aquestes iteracions es basen les diferents funcionalitats que s'han afegit i el nivell de complexitat que anaven adquirint els mòduls.

S'ha volgut programar cada mòdul a part de la resta, de tal manera que es pogués executar independentment abans d'incorporar-lo al projecte. Així es facilita començar a provar cada nova part del programa al marge de la resta. Aquest procés ha permès testejar molt sovint cadascun dels mòduls.

En el cas de l'arbre de comportament en C++ s'han programat les classes corresponents als nodes principals i executat programes de prova al microcontrolador, depurant per port sèrie en un terminal.

Més endavant s'han afegit altres nodes decoradors —que donen més opcions al programador— i finalment els nodes destinats a la programació amb múltiples fils, ja que suposaven fer canvis importants en la programació interna dels nodes tal i com s'ha explicat en l'apartat de desenvolupament.

En el cas de les proves amb els nodes multi-fil han estat generats programes semblants als usats en l'execució en sèrie, però replicant les tasques per ser executades en paral·lel.

L'apartat de l'entorn de desenvolupament té més mòduls. En el cas del panell de disseny s'ha començat amb un panell on arrossegar etiquetes. S'hi han anat afegint funcions fins completar-lo: enllaços entre objectes, gràfics o un editor intern per canviar el nom dels nodes. Tots els mòduls s'han anat incorporant al programa però òbviament s'han acabat de desenvolupar un cop integrats.

Altres mòduls que cal mencionar són: l'editor de text, el verificador de l'arbre de comportament, l'exportador i importador dels dissenys, el generador de codi o el compilador. Tots aquests elements poden ser provats de forma independent de la resta del programa amb paràmetres que no són més complexos que els d'un objecte Node, els de cadenes de caràcters o els d'un fitxer.

L'arbre de comportament que s'executa en el microcontrolador es relaciona amb l'arbre de comportament de l'entorn de desenvolupament a la fase de generació i compilació del codi font. Aquest codi font generat ha de tenir el mateix format que qualsevol codi compilable amb l'eina en línia que ofereix Mbed. Però tal com era

requerit, també ha de ser compatible amb el compilador local de l'IDE, que com ha estat esmentat, genera un binari directament executable al microcontrolador.

Proves amb l'IDE i execució en el microcontrolador

Per validar el funcionament del projecte es passa finalment per totes les fases del procés, des del disseny amb l'editor gràfic de l'IDE fins a la compilació i execució del programa. Han estat dissenyats diversos programes d'exemple per verificar que es pot construir qualsevol comportament amb els blocs desenvolupats.

La figura següent representa un circuit que té un polsador. Per defecte, el circuit és obert i el potencial llegit pel port *In* és 5V o un 1 lògic. En polsar el botó, es tanca el circuit i la resistència fa caure el voltatge canviant el valor de l'entrada digital a 0. Un led de la placa del microcontrolador indica l'estat del circuit.

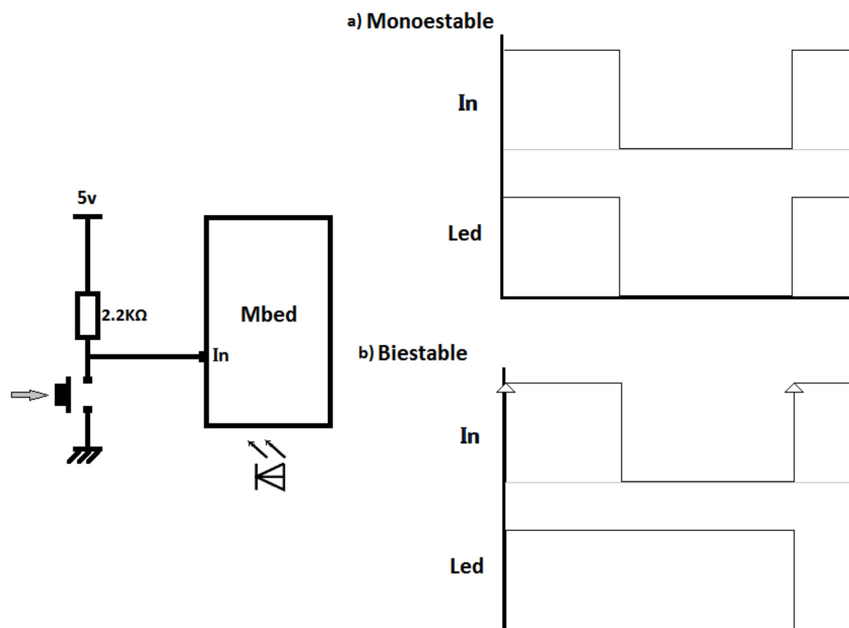



Figura 36. Circuit amb el polsador i gràfica de la sortida (Led) segons l'entrada (In).

Per aquest circuit han estat programats dos comportaments mitjançant l'entorn de desenvolupament del projecte:

El primer comportament (figura 35(a)) té com a funció obtenir un circuit monoestable. En aquest cas el led s'il·lumina o no en funció de l'entrada *In* instantàniament.

El segon comportament (figura 35(b)) té com a funció obtenir un circuit biestable i pot trobar-se en dos estats durant un temps indeterminat. El polsador, en aquest cas, serveix per canviar l'estat del circuit (engegat/apagat). Aquest canvi d'estat es determina detectant els flancs de pujada.

Els dos programes s'han dissenyat, programat i compilat amb l'entorn de desenvolupament i han estat verificats amb el microcontrolador mbed i el circuit descrit.

La figura 36 representa els arbres de comportament Monoestable (a) i Biestable(b). Hi ha molts dissenys diferents possibles i a més el programador pot decidir si donar més càrrega funcional als nodes o a la programació manual.

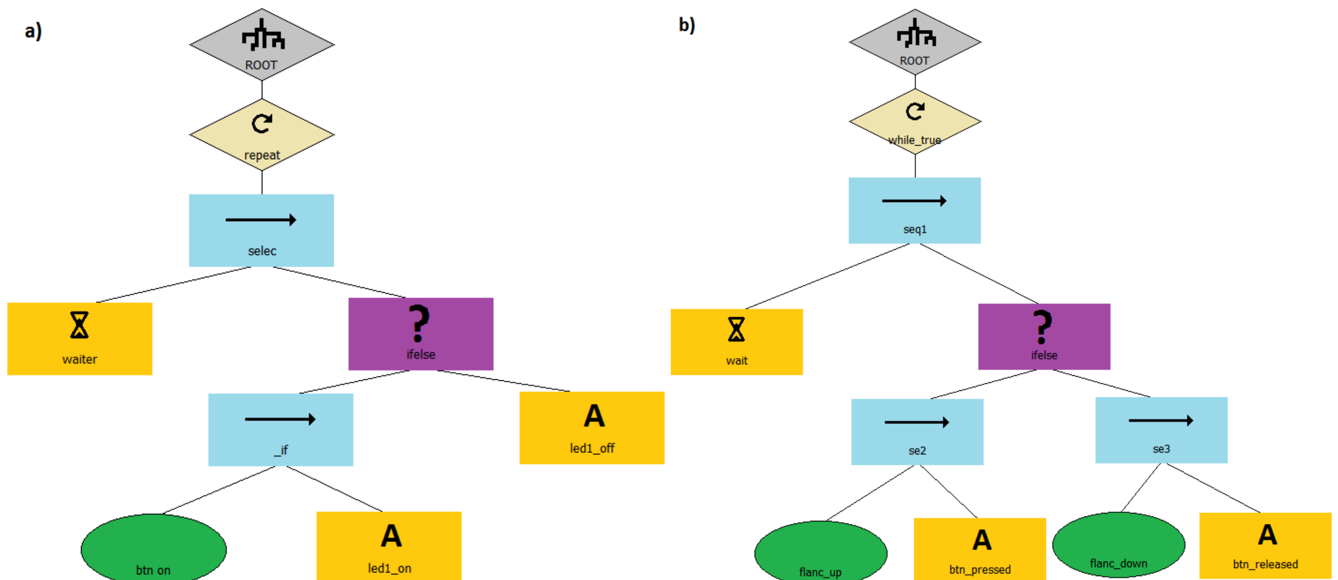



Figura 36. Diagrames del circuit monoestable(a) i biestable(b)

Per defecte, l'arbre de comportament no disposa d'estructures pròpies per desar informació. Aquest exemple usa una variable global per referir el led i una variable extra en el model biestable que representa l'estat del circuit (flanc). El node acció activa o desactiva el led i manté la informació sobre l'estat. La condició només verifica la pulsació del botó per desencadenar una acció o una altra.

Finalment, cal esmentar que el nombre reduït d'anotacions a la bibliografia és deu a que no he trobat referències i treballs previs tant específics com és el desenvolupament d'un entorn de disseny i programació, i menys encara d'una implementació d'un arbre de comportament en un hardware específic. Tot i això la referència principal en quant a la teoria emprada ha estat el llibre de Millington i Funge, Artificial Intelligence for Games.

# 7. Cronograma

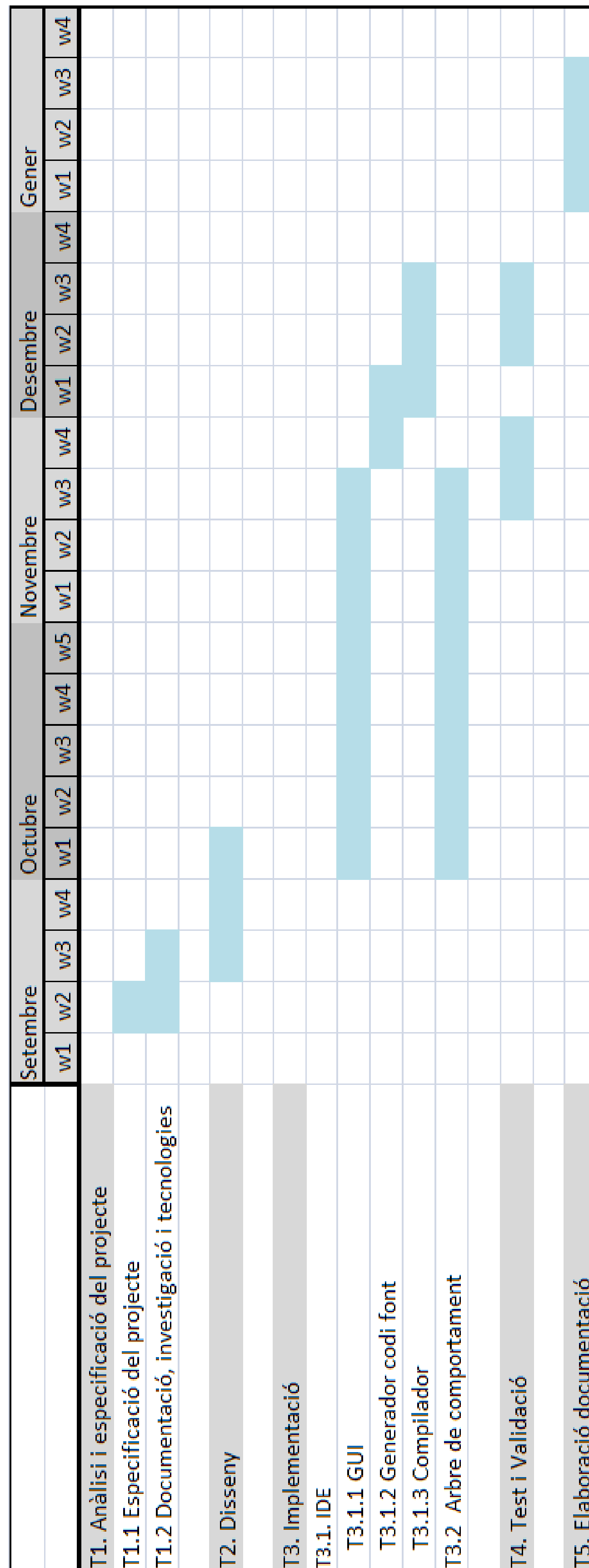



Figura 37. Cronograma del projecte.

# 8. Valoració econòmica del projecte

Inicialment valoro econòmicament el projecte segons els costos imputats als següents àmbits:

a) Hores destinades a les diverses tasques del projecte.

| Tasques | Hores |
|---|---|
| T1. Anàlisi i especificació del projecte | |
| T1.1 Especificació del projecte | 40 |
| T1.2 Documentació, investigació i tecnologies | 60 |
| T2. Disseny | 100 |
| T3. Implementació | |
| T3.1. IDE | |
| T3.1.1 GUI | 140 |
| T3.1.2 Generador codi font | 60 |
| T3.1.3 Compilador | 60 |
| T3.2 Arbre de comportament | 140 |
| T4. Test i Validació | 75 |
| T5. Elaboració documentació | 120 |
| Total | 795 |

Figura 38. Distribució de temps estimat per les tasques.

El temps destinat a cada tasca és aproximat ja que com es pot comprovar al diagrama de gantt(figura 37) algunes tasques s'han desenvolupat alhora. D'altres com el test s'encavalca amb la implementació i s'han de tenir en compte setmanes que al calendari i segons festius i mesos no són complertes.

b) Despeses generals entre les quals principalment tenim:

- Hardware (microcontrolador) 60€.
- Lloguer d'una oficina o sala de treball amb serveis integrats. 200€ mes per 5 mesos.
- Amortització equip informàtic. Preu de compra de 900€ entre 60 mesos per 5 mesos d'ús .
- El software que he utilitzat és lliure i no té costos de llicència.

La suma de costos dona el següent resultat:

| Concepte | Preu |
|---|---|
| Cost enginyer 795h*30€ | 23.850€ |
| Hardware (microcontrolador) | 60€ |
| Lloguer oficina | 1.000€ |
| Amortització equips | 75€ |
| **Total** | **24.985€** |

Figura 39. Taula de costos del desenvolupament del projecte.

L'import de la taula anterior correspon estrictament a costos suportats en el desenvolupament del projecte. La seva valoració de mercat dependria de l'acceptació que pogués tenir entre els seus clients potencials, tema que va més enllà de l'abast d'aquest projecte.

# 9. Conclusions

En aquest projecte s'ha desenvolupat, al voltant d'un tipus de diagrama, l'arbre de comportament per ser utilitzat com a eina que faciliti el disseny de programes destinats al control i  l'automatització de sistemes.

Amb l'objectiu de facilitar la generació del codi de programa  a partir del disseny en forma de diagrama s'ha creat una implementació d'un arbre de comportament amb el llenguatge de programació C++ per a un microcontrolador.

S'ha desenvolupat un entorn de treball gràfic o IDE per integrar el disseny i la programació de l'arbre de comportament.

L'entorn de desenvolupament ha estat dotat de les eines per generar el codi font del programa corresponent a l'arbre de comportament dissenyat. També s'hi ha incorporat un compilador.

Fent un ús conjunt de totes les eines s'han dissenyat, programat i generat codis de programa de test que han estat compilats i executats amb èxit en un microcontrolador.

L'IDE fa que l'arbre de comportament sigui la base al voltant de la qual giren totes les altres tasques del procés de desenvolupament. D'aquesta manera el pes del programa recau més sobre l'arbre(disseny) que si el conjunt de les parts haguessin estat dissenyades programades per separat.

# 10. Treball futur i millores

L'eina que s'ha desenvolupat en aquest projecte disposa de les funcionalitats bàsiques que permeten dissenyar programes destinats a un microcontrolador concret a partir d'un arbre de comportament.

Com a possibles millores en una nova iteració del projecte distingim les següents:

1) Pel que fa a l'arbre de comportament, podria ser dotat de més eines:

- Podrien ser afegits nous nodes, sobretot decoradors i nodes compostos. Aquests nodes podrien ser creats per realitzar alguna tasca concreta segons el tipus programa o comportament que volgués desenvolupar. En aquest cas els decoradors poden implementar fàcilment modificacions en el comportament dels fills. En el cas dels nodes compostos, dels quals pengen múltiples fills, poden implementar noves formes d'execució d'aquests fills. Per exemple es pot canviar l'ordre, la prioritat o les condicions segons les quals segueixen o en finalitzen l'execució.
- Una altra tècnica —l'ús del node *blackboard*— es pot fer servir a l'arbre de comportament per manipular variables locals en una de les seves branques. Aquest canvi però, suposaria incrementar la complexitat de la implementació en C++ per al microcontrolador a l'hora de manejar l'estructura de dades i controlar-ne l'ús segons la branca (*scope*).

2) Pel que fa a L'IDE que s'ha dissenyat:

- Actualment el programa té en compte els errors que l'usuari pot cometre en fer-ne ús. Com s'ha explicat a l'apartat d'implementació es fan comprovacions bàsiques sobre la correcta construcció de l'arbre en les diverses fases com dissenyar, desar i compilar el codi. Tot i això, una futura millora hauria de fer més estrictes aquests controls. Per exemple, és molt complicat realitzar un *parsing* del codi font dels nodes per tal de comprovar la validesa, no repetició o la correcta inicialització de les variables programades.
- En quant al codi de programa que l'IDE genera fins ara, és prou senzill per posar-lo en un sol fitxer de text. Seria interessant separar les classes generades i usar fitxers de tipus *header* (**.h*) per poder dissenyar comportaments integrables a altres programes.
- El compilador integrat fa ús de la configuració més senzilla. Donaria més versatilitat al programa trobar un compilador més portable que el que s'usa i configurar-lo per ser utilitzat en més sistemes operatius.

# 11. Annexos

### *Annex 1. Classe Paral·lel*

```
class Parallel : public CompositeNode {
    private:
        Queue<bool, 16> resQueue;
        Mutex res_mutex;
        int results;
        int consumed;
    public:
        Parallel() {
            results = 0;
            consumed = 0;
        }
        virtual bool run() {
            vector<Node*> v = getChildren();
            for (int i = 0; i < v.size(); ++i) {
                v[i]->run();
            }
            while(consumed < v.size()) {
                osEvent evt = resQueue.get();
                if (evt.status == osEventMessage) {
                    bool *returnValue = (bool*)evt.value.p;
                    if ( *returnValue == false) {
                        //Finalitzar fills
                        return terminate();
                    }
                    ++consumed;
                }
            }
            return true;
        }
        void setResult(bool* b) {
            res_mutex.lock();
            resQueue.put(b);
            res_mutex.unlock();
        }
        virtual bool terminate() {
            vector<Node*> v = getChildren();
            for (int i = 0; i < v.size(); ++i) {
                v[i]->terminate();
            }

            return false;
        }

};
```

## *Annex 2. Classe Acció*

```
class Action : public Node {
    private:
        Thread *_thread;
        bool _isThread;
        Parallel *_parent;
        static void threadStarter(void const *p) {
            Action *instance = (Action*)p;
            instance->runable();
        }
        void runable() {
            if(_isThread)
                _thread->signal_wait(START_THREAD);

            //Node Action code

            returnValue = true;
            //Now notify parent
            if(_isThread)
                _parent->setResult(&returnValue);
        }
    public:
        bool returnValue;
        Action(Parallel *parent=NULL) {
            if(parent != NULL) {
                _isThread = true;
            } else {
                _isThread = false;
            }
            _parent = parent;
            returnValue = false;
            if (_isThread)
                _thread = new Thread(&Action::threadStarter, this,
                                   osPriorityNormal,512);
        }
        virtual bool run() {
            if(_isThread)
                _thread->signal_set(START_THREAD);
            else
                runable();
            return returnValue;
        }
        virtual bool terminate() {
            //Finishing the thread OK
            if (_thread->get_state() == Thread::Inactive)
                return false;
            _thread->terminate();
            return false;
        }
};
```

## *Annex 3. XSD del fitxer d'exportació del diagrama*

```
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="BTree">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="node" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:byte" name="id"/>
              <xs:element type="xs:byte" name="nodetype"/>
              <xs:element type="xs:string" name="nom"/>
              <xs:element type="xs:byte" name="parent"/>
              <xs:element name="children">
                <xs:complexType mixed="true">
                  <xs:sequence>
                    <xs:element type="xs:byte" name="child"
                             maxOccurs="unbounded" minOccurs="0"/>
                  </xs:sequence>
                  <xs:attribute type="xs:byte" name="numchildren"
                             use="optional"/>
                </xs:complexType>
              </xs:element>
              <xs:element name="location">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:short" name="x"/>
                    <xs:element type="xs:short" name="y"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element type="xs:string" name="data"/>
              <xs:element type="xs:byte" name="numrepeticions"
                             minOccurs="0"/>
              <xs:element type="xs:short" name="mswait"
                             minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

# 12. Bibliografia

[vii][viii][ix][x][xi][xii][xiii][xiv]

Les referències anteriors [VII-XI] corresponen a documentació en general que m'ha servit com a font d'informació per desenvolupar i obtenir idees per la implementació de l'arbre de comportament. Les tres últimes referències m'han servit d'inspiració pel disseny de la interfície gràfica(XII, XIII) i documentació de les llibreries utilitzades(XIV a part de IV).

---

[i] Millington, I. & Funge, J. (2009). Artificial Intelligence for Games. (p334) Taylor & Francis.
[ii] Mbed. http://developer.mbed.org/platforms/mbed-LPC1768 [Accedit: 1-10-2014]
[iii] ARM. http://www.keil.com/support/man/docs/armccref/armccref_babcfbgj.htm [Accedit: 1-10-2014]
[iv] RSyntaxTextArea. http://fifesoft.com/rsyntaxtextarea [Accedit: 22-11-2014]
[v] Gcc4Mbed. https://github.com/adamgreen/gcc4mbed [Accedit: 4-11-2014]
[vi] XmlSchema. http://www.w3.org/XML/Schema [Accedit: 21-12-2014]
[vii] Klöckner, Andreas. "Behavior Trees for UAV Mission Management." GI-Jahrestagung. 2013.
[viii] Ogren, Petter. "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees." AIAA Guidance, Navigation and Control Conference, Minneapolis, Minnesota. 2012.
[ix] http://www.gamasutra.com. Behavior trees for AI: How they work
[x] http://aigamedev.com
[xi] http://bt.multi-threading.com/concept/bt/
[xii] https://github.com/cmonty/brainiac.
[xiii] jgraph.com
[xiv] http://docs.oracle.com