dFSM: Diseño de Sistema Empotrados con Autómatas Distribuidos

Andreu Moreno
Escola Universitària Salesiana de Sarrià (EUSS).
St. Joan Bosco, 74.
08017 Barcelona.
www.euss.es
amoreno@euss.es

Joan Valduvieco
LAIGU, SCCL
www.laigu.net
joan@laigu.net

Resumen

El proyecto dFSM, presentado en este artículo, pretende realizar una aportación al diseño de los sistemas implementables mediante un conjunto de autómatas finitos. Los sistemas empotrados (embedded) acostumbran a poderse modelar como máquinas de estados finitos alimentadas por eventos externos. El proyecto dFSM aborda su diseño dentro del mundo del software libre, de manera que sea lo más simple posible. El usuario-programador dispone de una interfaz que, de forma clara y simple, le permite codificar la lógica del sistema en una serie de autómatas, los cuales se comunican mediante un mecanismo de eventos basado en publicación - subscripción. El proyecto se ha realizado bajo licencia LGPL y se gestiona en el portal http://www.sourceforge.net.

1. Introducción

En general los sistemas pueden clasificarse en Transformadores y Reactivos [1][2]. Los Transformadores son aquellos que pueden describirse con una función de entrada/salida, en cambio los Reactivos se caracterizan por actuar en respuesta a estímulos (internos o externos). Los sistemas empotrados son mayoritariamente Reactivos [3], y la experiencia demuestra que es difícil describir su comportamiento de forma clara, realista y formal. Para facilitar el diseño se han realizado varias contribuciones; las más significativas son los diagramas de estados de Harel [2], los diagramas de estados en UML [4] y la Programación Cuántica [5]. Todas estas aportaciones se basan en mejorar el modelo clásico de las máquinas de estados. El proyecto dFSM recoge estas mejoras y proporciona una infraestructura que permite trabajar con estos sistemas bajo software libre.El artículo empieza con la descripción de los conceptos básicos sobre las máquinas de estados clásicas como antecedentes del proyecto, a continuación se explica la estructura de un sistema basado en dFSM y se presenta un ejemplo de aplicación. Finalmente se comentan los trabajos relacionados, las conclusiones y perspectivas de futuro.

2. Máquinas de estados finitos

La aproximación clásica a les máquinas de estados finitos conduce al autómata de Mealy y al autómata de Moore. El autómata de Moore asocia acciones a los estados; este hecho implica que el comportamiento del sistema sólo depende del estado actual. En cambio, el autómata de Mealy asocia las acciones a las transiciones entre estados, lo cual supone que su comportamiento dependa además del estado actual, del evento que acaba de llegar. En general cada máquina de estados finitos puede modelarse mediante uno de los autómatas clásicos, aunque la implementación de Moore acostumbra a tener más estados.Cualquier proceso complejo de diseño requiere dividir el problema en partes funcionales y afrontarlo primero des de una óptica abstracta y general hasta una de más concreta y detallada, pasando por sucesivas representaciones del modelo con más complejidad a medida que se aproximan al sistema estudiado. Las máquinas de estados clásicas están basadas en un diseño talmente plano, y este hecho que no facilita una división lógica en partes, y además, la experiencia demuestra que a medida que el nombre de estados crece en un modelo plano, la complejidad lo hace

exponencialmente. Para superar las deficiencias del modelo clásico, David Harel [2] desarrolló el concepto de Statecharts, diagramas de estados, que aportan la introducción de la jerarquía a los diagramas de estados. Conceptualmente supone la introducción de estados que contienen en su interior otro diagrama de estados completo. En la Figura 1 muestra un primer diagrama de estados. Los estados se identifican con rectángulos redondeados y las transiciones con flechas acompañadas del evento que las desencadena. Se observa como el estado S1 contiene los estado S11 y S12, lo cual implica que los estados hijos heredan las transiciones del padre (en este caso la asociada al evento E2). El hecho de añadir una nueva dimensión a los diagramas de estados comporta que los diseños sean más simples y mejor estructurados. Otra de las contribuciones destacadas de Harel son los and-states que permiten tener concurrencia en la máquina de estados. Un and-state consiste en un estado padre que tiene más de un diagrama de estados en su interior. Estos sub-diagramas tienen un estado propio y se ejecutan de forma concurrente. La base conceptual sobre diagramas de estados introducida por Harel fue suficientemente importante como para que el OMG (Object Manegement Group) la utilizase como punto de partida para especificar uno de los diagramas de comportamiento en el lenguaje de modelado UML[4].

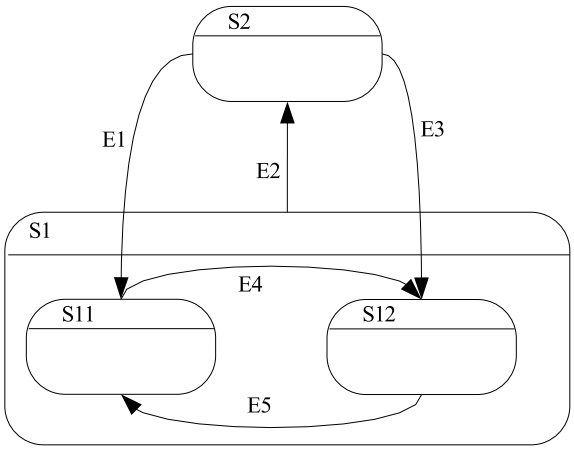

FIGURA 1: *Jerarquía en los diagramas de estados*

Recientemente se ha introducido el concepto de Programación Cuántica[5] como base para implementar los diagramas de estados en lenguaje C/C++. Su aparición ha supuesto romper la idea preconcebida de que los diagramas de estados sólo poden utilizarse en la especificación de los sistemas, pero no de una forma sistemática en la fase de codificación. Samek, el autor de la Programación Cuántica, madura el concepto de herencia de comportamiento[5], previamente introducido por Harel[2], como evolución natural de la herencia en la programación orientada a objetos. La herencia de comportamiento especifica que, en un diagrama de estados jerárquico, los estados hijos heredan las transiciones asociadas a su estado padre, es decir, el comportamiento.

3. Proyecto dFSM

El proyecto dFSM (Distributed Finite State Machine) pretende desarrollar una plataforma que permita modelar un sistema como un conjunto de autómatas finitos (Figura 2). Estos autómatas están distribuidos entre diferentes procesos en una misma máquina, o en varias si se dispone de un mecanismo de comunicación fiable entre ellas.El desarrollo del proyecto se realiza utilizando el Sistema Operativo GNU/Linux, aunque el hecho de que se haya implementado en C++ permite portarlo a otras plataformas con facilidad. Cada autómata que utilice dFSM tiene entidad de tarea del sistema operativo, lo cual supone disponer de un espacio de memoria aislado del resto y una prioridad propia si es necesario. El usuario final utiliza una interfaz orientada a objetos realizada en C++ que simplifica la fase de codificación y fomenta el paralelismo entre el diseño formal y la implementación. Inspirado en el bus de comunicaciones industriales CAN [6], se ha desarrollador un mecanismo de comunicaciones (Figura 2) basado en el intercambio de eventos por publicación - subscripción, donde cada mensaje (evento) es identificado por la información que transporta y no por el destinatario. En proceso de arranque los autómatas examinan todos los eventos a los que tienen asociadas transiciones y se subscriben automáticamente. La recepción de estos eventos es lo que desencadena la evolución del estado de los autómatas, siempre y cuando el estado actual del autómata tenga una transición asociada al evento entrante.

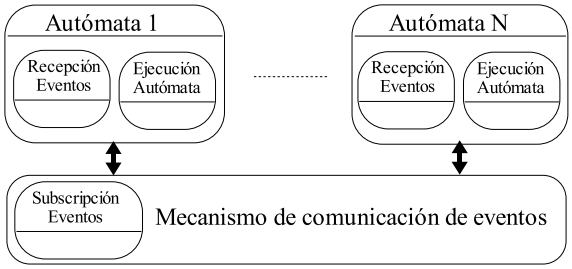

FIGURA 2: *Autómatas distribuidos con mecanismo de comunicación de eventos*

El proyecto dFSM parte de la base conceptual expuesta, y tiene las siguientes características:

- Herramienta de modelado de sistemas Reactivos. El proyecto dFSM pretende ofrecer una solución simple a los sistemas alimentados por eventos. El programador no tiene que preocuparse de la implementación de los autómatas, se establece un procedimiento directo para pasar del modelo formal al código. Paralelamente se está desarrollando una herramienta gráfica que permite simplificar aun más la codificación.

- Estructura jerárquica. Los autómatas jerárquicos han sido uno delos avances más significativos en el área de los autómatas de estados

finitos. Añaden otra dimensión a los autómatas clásicos, hecho que permite un diseño más simple y claro.

- Comportamiento histórico y estado por defecto. Posibilidad de seleccionar que un estado padre tenga un comportamiento histórico, es decir, que recordará el estado a la salida para recuperarlo a la vuelta. Otra posibilidad es especificar el estado hijo que se seleccionará siempre que se entre al estado padre.

- Transiciones. Los cambios entre estados se denominan transiciones, son provocadas por la llegada de un determinado evento.

- Herencia de comportamiento. Los estados hijos heredan el comportamiento (transiciones) del su estado padre. Esta característica da sentido a la estructura jerárquica.

- Mecanismo de intercambio de eventos. Se dispone de un mecanismo de comunicación de eventos basado en publicación -subscripción. Los autómatas se registran previamente a los eventos que desean recibir (que obviamente serán los asociados a las transiciones del autómata). Paralelamente se especifica una prioridad asociada a los eventos que permite ordenarlos en recepción en el caso de acumulación.

- Eventos diferidos y eventos desestimados. Cuando llega un evento que no tiene ninguna transición registrada en el estado actual, el comportamiento por defecto es desestimarlo, y por tanto se pierde. En ocasiones puede ser interesante que este evento se difiera para ser procesado posteriormente. Los eventos diferidos permiten especificar este comportamiento.

- Procesado secuencial y atómico de los eventos. En dFSM, como en la mayoría de implementaciones, cuando el sistema está ejecutando unarespuesta a un evento (transición), no se procesa ningún evento hasta que no se encuentre estable en un estado. Esta naturaleza no-preferencial (non-preemptive) es necesaria tenerla presente en sistemas con requerimientos de tiempo real muy estrictos, donde laparte más restrictiva deberá implementarse como autómatas perfectamente identificados, y sería entonces el propio Sistema Operativo el que garantiza la respuesta. El hecho de distribuir la aplicación en diferentes autómatas hace posible que la responsabilidad derespuesta en tiempo real se delegue al sistema operativo, si el diseñador ha aislado correctamente las partes del sistema más críticas.

- Concurrencia. los and-states de Harel se han introducido de forma diferente al proyecto dFSM. El programador define diferentes autómatas los cuales se ejecutarán de forma concurrente y se comunican con el mecanismo de eventos. Esta aproximacióntiene un cierto paralelismo con los sistemas distribuidos como CORBA, donde los objetos tienen su propio hilo de ejecución(denominados Actores) y intercambian información a través de interfaces públicas, la cuales se subscriben previamente.

- Basada en objetos. Tanto a nivel interior como la interfaz que se ofrece al programador están orientadas a objetos. El lenguaje de programación utilizado es C++.

- Acciones. El programador final puede asignar acciones, código a ejecutar, tanto en los estados como en las transiciones, por lo tanto, se soportan los autómatas de Mealy y los de Moore. Referente a los estados, se puede definir una acción al entrar y otra al salir. Esta dualidad en las acciones asociadas a los estados es similar al papel de constructor y el destructor en la creación de un objeto. Es por eso que cada transición supone la ejecución ordenada de las acciones de salida de los estados que se abandonen dentro de la estructura jerárquica, a continuación la acción asociada a la transición y finalmente las acciones de entrada a los estados que se encuentren en el camino des de el estado origen hasta el estado destino.

- Motor de ejecución. Se ha desarrollado un motor de ejecución que controla la evolución de cada autómata y se encarga de recibir los eventos a que previamente se habrá registrado el autómata. Desde un punto de vista estructural (Figura 2), cada autómata dispone de dos threads, uno dedicado a recibir eventos y otro de ejecutar el código de entrada y salida de los estados y de realizar las transiciones. Entre ellos se comunican mediante una estructura de datos común convenientemente protegida con mecanismos de exclusión mutua.

- Estructura interna de datos. Dentro de la información interna cabe distinguir entre la que es compartida por todos los autómatas y la que es propia de cada uno. En el primer caso se ha optado por situar en memoria compartida una tabla que registra los eventos a los que está subscrito cada autómata. Esta información es utilizada en cada ocasión que se procede a enviar un evento para obtener los autómatas que están interesados en el mismo, con los que se establece una conexión mediante sockets UNIX. En el segundo caso, cada autómata dispone de una lista con los diferentes estados y transiciones que será utilizada para determinar en todo momento cual es el siguiente estado en función de los eventos entrantes.

Finalmente el programador dispone de una interfaz para codificar los autómatas que está contenida en unas librerías compartidas, a las que se ha de enlazar la aplicación. El hecho de haber seleccionado una licencia de tipo LGPL (GNU Lesser General Public License), permite que se utilice en aquellas aplicaciones que requieran tener licencia propietaria. La gestión se lleva a cabo en el portal de software libre "sourceforge.net", hecho que supone disponer de todos los servicios que ofrece este portal: CVS, listas de correo, foros, bugtraq,.... Se puede acceder desde http://www.sourceforge.net/projects/dfsm o bien http://dfsm.sourceforge.net.El proyecto dFSM nace como resultado de la consultoría realizada por la escuela universitaria EUSS [7] y la empresa LAIGU [8] a la empresa CIRSA INTERACTIVE [9], frente la necesidad de disponer de una plataforma para diseñar sistemas empotrados con el paradigma de los autómatas distribuidos. El proyecto fue ideado y diseñado por los autores de este artículo, y actualmente se mantiene voluntariamente fuera del ámbito de la relación empresarial inicial.

4. Aplicación ejemplo

Para mostrar la utilización del proyecto dFSM se propone la implementación de un robot rastreador de línea. Este tipo de robots siguen una línea negra pintada en el suelo. Para ello disponen de tres fotosensores reflexivos en la parte delantera para orientarse. Mediante la información suministrada por los sensores deciden actuar individualmente sobre el movimiento de cada una de las ruedas principales, las cuales pueden girar hacia adelante, atrás o estar paradas. En la parte posterior se coloca una rueda giratoria libre. La aplicación que controla el movimiento del robot se ha dividido en cuatro autómatas: `Rueda_derecha`, `Rueda_Trasera`, `Control` y `Sensores`. Cada uno de estos autómatas tiene una serie de estados. En la Tabla 1 se describen. Los autómatas se comunican mediante el mecanismo de eventos. En la Tabla 2 se presentan los diferentes eventos, remarcando quien los genera y qué autómatas estarán registrados.

Cada uno de los autómatas se modeliza en un diagrama de estados dónde se representan los estados, las transiciones y los eventos las provocan. También se le añade de forma esquematizada los eventos que se generen. En la Figura 3 muestra el diagrama del autómata de los sensores. En este caso se trata de comprovar de forma periódica el estado de los sensores y generar los eventos asociados. Esta tarea se realiza en el método `Entry`. Adicionalmente se genera el evento `E_SEspera`, al cual está registrado el propio autómata y que provoca una transición sin cambio de estado pero si que se ejecuta la acción asociada a la transición, en esta caso un tiempo de espera.

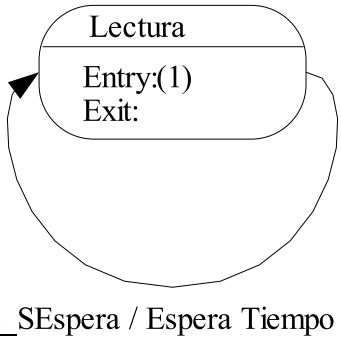

FIGURA 3: *Diagrama de estado del autómata Sensores*

[1] El diagrama de estados del autómata de unos de los motores se representa en la Figura 4. Destaca el estado padre `Mover` que contiene los estados `Adelante` y `Atras`. Esta jerarquía permite simplificar laespecificación de la transición que se dirige al estado `Paro`, pues es heredado por los dos estados hijos.Finalmente en la Figura 5 muestra el diagrama del autómata `Control`. Se definen una jerarquía con tres estados de movimiento. También en este caso se manifiesta que ha jerarquía permite que los tres estados hijos heredan la transición hacia el estado `Paro` con el evento `E_Paro`. La transición asociada al evento `E_Continua` se dirige al estado padre `S_Mover`, es decir, no se especifica el estado hijo destino, pero se ha puesto un H circunscrita que indica que el estado final es el último ejecutado y en su defecto el estado `Adelante` (flecha que llega des de el interior). En cuanto a la generación de eventos, cada uno de los estados de movimiento genera los eventos necesarios a su entrada para que los autómatas de las ruedas muevan el robot en la dirección deseada. Por ejemplo, el estado `Derecha` genera en la acción de entrada los eventos `E_AdelanteIzquierda` y `E_AtrásDerecha`, y a la salida `E_ParoIzquierda` y `EparoDerecha`.

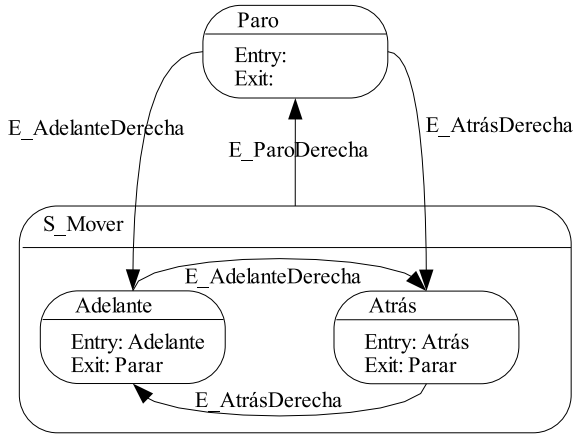

FIGURA 4: *Diagrama de estados de la rueda derecha*

Finalmente se presenta el código incompleto de la aplicación donde básicamente se pretende mostrar la

[1] Lectura de los sensores, generar los eventos según su estado y `E_SEspera`

Autómata	Estado	Descripción
Rueda_derecha	Parado	Rueda parada
	Mover	Estado padre de los siguientes
	Adelante	Rueda moviéndose hacia adelante
	Atrás	Rueda moviéndose hacia atrás
Rueda_izquierda	Parado	Rueda parada
	Mover	Estado padre de los siguientes
	Adelante	Rueda moviéndose hacia adelante
	Atrás	Rueda moviéndose hacia atrás
Control	Parado	Robot parado
	Mover	Estado padre de los siguientes
	Adelante	Robot moviéndose hacia adelante
	Atrás	Robot moviéndose hacia atrás
	Derecha	Robot girando hacia la derecha
	Izquierda	Robot girando hacia la izquierda
Sensores	Lectura	Estado Lectura sensores
	Espera	Espera siguiente lectura

Cuadro 1: Autómatas y estados

Evento	Generador	Registrados	Comentario
E_SDerecha1	Sensores	Control	Sensor derecha negro
E_SCentro1	Sensores	Control	Sensor centro negro
E_SIzquierda1	Sensores	Control	Sensor izquierda negro
E_Paro		Control	Parar el movimiento
E_Continua		Control	Continuar el movimiento
E_SEspera	Sensores	Sensores	Espera lectura sensores
E_ParoDerecha	Control	Rueda_Derecha	Parar rueda derecha
E_AdelanteDerecha	Control	Rueda_Derecha	Adelante rueda derecha
E_AtrasDerecha	Control	Rueda_Derecha	Atrás rueda derecha
E_ParoIzquierda	Control	Rueda_Izquierda	Parar rueda izquierda
E_AdelanteIzquierda	Control	Rueda_Izquierda	Adelante rueda izquierda
E_AtrasIzquierda	Control	Rueda_Izquierda	Atrás rueda izquierda

Cuadro 2: Eventos

definición de los estados y las transiciones de únicamente el autómata de la rueda derecha. En primer lugar se declaran los estados como clases que herendan de las clases bases que ofrece la librería dFSM. Observar que se puede definir una acción que se ejecutará al entrar el estado y otra al salir como métodos de la estructura. De hecho son métodos virtuales puros en la clase base `tState`. A continuación se declaran las transiciones con el mismo mecanismo y también con una acción asociada. Finalmente en la función `main` es donde se define el objeto que contendrá el autómata, se definen los estados y transiciones, se añaden al objeto autómata y empieza la ejecución. Para cada autómata se realiza un código similar de forma que al final se disponen de 4 ejecutables.

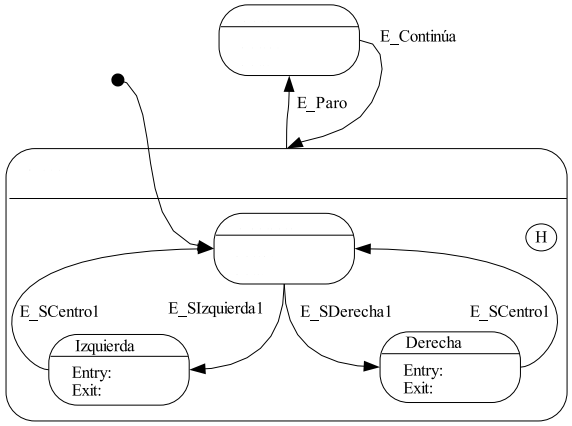

FIGURA 5: *Diagrama de estados del control*

```
struct tState_Mover :public tState
{
    void entry_func(){ /* ... código */}
    void exit_func(){ /* ... código */}
};

struct tState_Adelante :public tState
{
    void entry_func(){ /* ... código */}
    void exit_func(){ /* ... código */}
};
/*...*/
struct tTransition_Paro_Adelante:public tTransition
{
    void function(){ /* ... código */}
};
/*...*/
int main (int argc, char* argv[])
{
    tFSM *fsm; tState *s0, *s1; textttransition *t0;
    fsm  = new fsm("Nombre autómata",0);
    s0 = new tState_Mover();
    s0->set_id(0); // Identificador del estado
    fsm->add_state(s0);
    s1 = new tState_Adelante();
    s1->set_id(1);
    s1->set_parent(s0);
    fsm->add_state(s1);
/*...*/
    t0=new  textttransition_Paro_Adelante ();
    t0->set_source(1);
    // estado destino, suponemos que Paro = 2
    t0->set_target(2);
    t0->set_event(0);
    fsm->add_transition(t0);
/*...*/
    // Finalmente ejecutamos el autómata
    fsm->run();
}
```

5. Trabajos relacionados

Tal como ya se ha mencionado, Harel[2] revolucionó la teoría de los autómatas con la inclusión de la jerarquía y la concurrencia. Toda esta base se ha materializado en Statemate[10], aplicación que permite el análisis, diseño, y la posterior generación automática de código para sistemas reactivos. Es un software con amplias prestaciones , es una referencia en su sector pero totalmente propietario.Los autómatas forman parte también del lenguaje de modelado UML como base para representar modelos de comportamiento de los sistemas. En esta caso, el objetivo de las diferentes herramientas para trabajar con UML es mucho más amplio y genérico que el diseño de sistemas reactivos. Muchas de herramientas permiten como paso final generar código del sistema modelado y podrían usarse como base para diseñar este tipo de sistemas. Buenos ejemplos son Rhapsody de I-Logic , Rational de Rational Software Corp. y ObjectGeode de Telelogic. En todos los casos productos propietarios. En el campo de software libre, la alternativa para modelar con UML es Umbrello, paquete con prestaciones similares a los anteriores. La problemática de esta implementación reside en que se han concebido con una óptica más amplia en cuanto al tipo de sistema a diseñar, lo cual se manifiesta en una falta de eficiencia en el código generado para el caso concreto de sistema reactivos.Samek[5] presenta una aproximación distinta. En este caso no se trata de un paquete, sino de una infraestructura (denominada Quantum Framework), una librería, que permite una programación sistemática de autómatas distribuidos. En este sentido, la aproximación es similar a la de dFSM con un mecanismo de comunicación basado en eventos, pero con marcadas diferencias estructurales en la codificación de los estados y los autómatas. Por otro, aunque ofrece el código fuente, este está bajo licencia propietaria.En el campo del software libre, la aproximación más destacada es el proyecto SMC (State Machine Controller)[11] que lo plantea como un precompilador, en cual mediante un pseudo- lenguaje traduce la especificación de un autómata en código fuente C++. Por lo general soporta la mayoría de la funcionalidades pero con un aproximación distinta a la jerarquía, la cual está basada en equiparar un sub-diagrama de estados a una subrutina. Tampoco no realiza un tratamiento explícito de la concurrencia.También de la mano del software libre pero en este caso en el mundo

de la robótica y el control en general, dos proyectos se están consolidando. Por un lado MatPLC[12] está creando un infraestructura para poder diseñar un sistema como un conjunto de módulos, cada uno de ellos implementado como un PLC (Programmable Logic Controller, dispositivo de control utilizado ampliamente en el campo de la automatización industrial que se caracteriza por la simplicidad de su programación). Por otro lado el proyecto OROCOS[13] aborda también el diseño de un sistema de control pero en este caso con una plataforma basada en componentes inspirada en CORBA.El proyecto dFSM, presentado en este trabajo, aporta la base de los autómatas de las otras aproximaciones y además aborda el tema de la concurrencia de una forma singular hasta ahora. Debido a que de forma natural los programadores de sistemas empotrados modelan las aplicaciones como un conjunto de procesos más o menos independientes. dFSM añade la posibilidad de que cada uno de esos procesos puedan ser diseñados como una autómata de una forma clara y sencilla. Además se ofrece una infraestructura de comunicaciones basada en eventos por subscripción - publicación.

6. Conclusiones

En este artículo se ha presentado el proyecto dFSM como plataforma de programación que permite simplificar y sistematizar el diseño de sistemas basados en autómatas distribuidos. Se han presentado los antecedentes que han motivado el trabajo y se ha descrito la solución adoptada para dFSM. Finalmente se ha visto un ejemplo que pone de manifiesto la simplicidad de codificación y una comparativa de los trabajos existentes.Actualmente la librería tiene un nivel de desarrollo suficientemente estable como para ser considerada una alternativa de diseño en muchos proyectos de sistemas empotrados. Se sigue trabajando para mejorar la estabilidad y enriquecer las prestaciones. En primer lugar se la está dotando de un sistema de depuración propio, un sistema que permite tanto monitorizar la evolución de cada un de los autómatas como incidir en el su funcionamiento con la inserción de eventos. En segundo lugar se quiere disponer de un supervisor que se encargue de lanzar los diferentes autómatas de una aplicación y después supervisar su estado de salud, de tal forma que sea capaz por ejemplo de volverlos a lanzar si es necesario. Finalmente se está desarrollando una herramienta gráfica que permitirá realizar la fase de diseño en un entorno gráfico y posteriormente generar un esqueleto de los diferentes autómatas.

Referencias

[1] B.P. Douglass, 1998, *State Machines and Statecharts. Pts 1 and 2.* , http://www.quantum-leaps.com/resources.articles/Douglass01.pdf

[2] D.Harel, 1987, *Statecharts: A Visual Formalism for Complex Systems*, http://citeseer.nj.nec.com/harel87statecharts.html

[3] R.J. Wieringa, 2003, *Design Methods for Reactive Systems*, Yourdon, Statemate and the UML, publisher Morgan Kaufmann , ISBN 1558607552

[4] OMG, 2001, *OMG Unified Modeling Languaje Specification 1.4*

[5] M. Samek, 2002, *Practical Statecharts in C/C++*, publisher CMP Books , ISBN 1578201101

[6] CAN Bus, www.can-bus.com

[7] Escola Universitària Salesiana de Sarria (EUSS), www.euss.es

[8] LAIGU,SCCL, www.laigu.net

[9] CIRSA INTERACTIVE Corp. Terrassa, www.cirsa.com

[10] D. Harel and P Michal, 1998, *Modeling Reactive Systems with Statecharts, The STATEMATE Approach*, publisher McGraw Hill Text , ASIN 0070262055

[11] Proyecto CFSM, www.sourceforge.org/projects/cfsm

[12] MatPLC, mat.courceforge.net

[13] OROCOS Project, www.orocos.org