dFSM: Finite State Machines for Embedded Systems

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

Joan Valduvieco
LAIGU, SCCL
Cirsa Interactive Copr., Spain
joan@laigu.net

Abstract

Embedded systems usually can be modeled as finite state machines fed by events, formally known as reactive systems. The dFSM project aims to improve the development of systems that can be implemented with an automata set. The project starts from classical automatas, Moore and Mealy, Harel[1] hierarchical statecharts and Samek[2] Quantum Framework. A framework has been designed to allow to execute every automata as an independent user process that exchanges events using a mechanism based on publication / subscription. The final user has a simple C++ interface that makes the codification process more systematic so he can concentrate completely on modeling phase. The dFSM project is designed in the open source world over GNU/Linux, it has been released under LGPL license and is managed in http://www.sourceforge.net/projects/dfsm

1 Introduction

Traditionally Embedded systems have been developed without any formal methodology, which ends in a maintenace nightmare. In general, systems can be classified in Transformer and Reactive [1][2]. Transformer systems are those that can be described with an input/output function, on the other hand Reactives are characterized to act in answer to stimulus (internal or external). The embedded systems are mostly reactive [3], and experience demonstrates that it is difficult to describe its behavior in a clear, realistic and formal way. A few contributions have been done to formalize their design; the statecharts of Harel [2], the statecharts of UML [4] and the Quantum Programming [5] highlights among the most significant. All these contributions are based on improving the classic pattern of state machines. The project dFSM picks up these improvements and provides an infrastructure to work with these systems within an open source environment.

The paper begins with the description of the basic concepts of classical state machines as the basics of the project, hereinafter the structure of dFSM based systems is explained and an example is presented. Finally the related works, the summations and perspectives are commented.

2 Finite State Machines

In the classical approach finite states machines are the Mealy and Moore automatas. The Moore automata associates actions to states; this fact implies that the system behavior only depends on the current state. On the other hand, Mealy automata associates actions to the state transitions, which supposes that its behavior depends on the current state and the arrived event. In general each finite state machine can be modeled by means of one of the classic automatas, although the Moore implementation often have more states.

Any complex design process requires to divide the problem in functional parts and confront it from an abstract view to a detailed one. It is carried out by successive representations that gain in complex-

ity as approach to the studied system. The classical state machines are based on a completely flat design, this fact doesn't help a logical division in parts, and also, experience demonstrates that as the number of states grows in a plane model, the complexity does it exponentially.

To overcome the deficiencies of the classical pattern, David Harel [2] developed the concept of Statecharts, which added the hierarchy to the state diagrams. Conceptually it supposes the introduction of states that contain another state diagram inside. Figure 1 shows a simple hierarchical state diagram. The states are identified with rounded rectangles and the transitions with arrows with the event that triggers them. As can be observed that the state S1 (parent state) contains the states S11 and S12 (son states). In case the state machine is in anyone of son states, the arrival of event E2 would suppose exit from current state, exit its parent state S1 and enter state S2. This example shows the inheritance of transitions from parents to sons. This fact adds a new dimension to the state diagrams, and allows simpler and structured designs. Another outstanding contribution from Harel are and-states (also named orthogonality) which allows concurrence in state machines. An and-state consists on a parent state that has more than one diagram inside. Each of these subdiagrams have its own state. The conceptual base on diagrams introduced by Harel was used by the OMG (Object Management Group) as a departure point to specify one of the behavior diagrams in the UML[4].

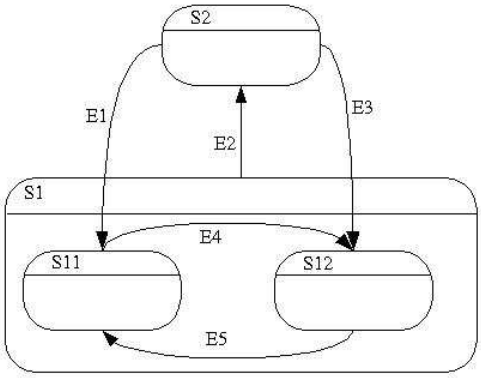

FIGURE 1: *Hierarchical state diagram*

Recently the concept of Quantum Programming [5] has been introduced as a base to implement the state diagrams using C/C++ language. Its appearance has supposed to break the preconceived idea that the state diagrams only can be used in the system specification process, but not as a systematic coding way in the coding phase. Samek, the author of the Quantum Programming, matures the behavioral inheritance concept [5], previously introduced by Harel[2], as natural evolution of the inheritance in the object oriented programming. The behavioral inheritance specifies that, in a hierarchical state diagram, the son states inherit the transitions associated to its parent, that is to say, the behavior.

3 dFSM Project

The dFSM project (Distributed Finite State Machine) aims to develop a platform that allows modeling a system like a group of automatas (Figure 2). These automatas are distributed among different processes in the same machine, or in several if we have reliable communication mechanism among them.

The project has been developed under GNU/LINUX, using C++ which eases porting to other platforms. Each dFSM automata has system process entity implying an isolated memory space and own priority when necessary. The end user has a C++ object oriented interface which gives access to the classes that modellate the machine, the states and transitions. Inspired by the industrial communications bus CAN [6], a communications mechanism has been developed (Figure 2). This communication mechanism is based on event exchange by means of publication - subscription. Each message (event) is identified by the information that transports and not by its destination. At startup every automata automatically subscribes to the events that triggers any of its transition. When an event is received by the automata the transitions assiciated to the current state are examined. If any of these transitions is triggered by the arriving event the transition process to the target state is carried out.

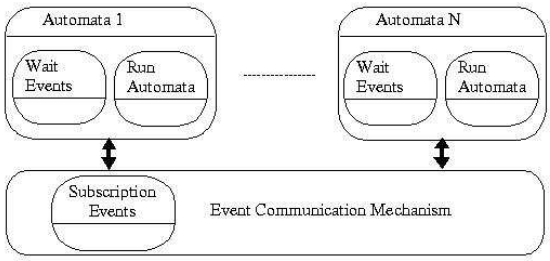

FIGURE 2: *Distributed automatas with event communication mechanism*

The dFSM project starts from the exposed conceptual base, and has the following characteristics:

- Reactive system modeling tool. The dFSM project seeks to offer a simple solution to the systems fed by events. The programmer doesn't have to worry about the automatas implementation, a direct procedure has been

setup to formal model pattern to code. Also we are now developing a graphic tool that will allow to simplify more the code phase.

- Hierarchy support. The hierarchical automatas have been one of the most significant advances in the this area. They add another dimension to the classic automatas, that allows a simpler and clearer design.

- Historical behavior and default state. The user has the possibility to select whether a parent state has a historical behavior, that is to say the current son state is remembered when the parent is exited and recovered when reentering. Another possibility is to specify the son state that will be selected whenever the parent state is entered.

- Transitions. The interstate changes are denominated transitions, they are caused by the arrival of certain event. Transitions can have code associated to them.

- Behavior inheritance. The son states inherit the behavior (transitions) of their parent state. This characteristic gives sense to the hierarchical structure.

- Event exchange mechanism. A communication mechanism has been developed a communication mechanism based on publication - subscription. The automata registers to the events to which a transition is associated. Also a priority associated to the events is specified that allows to order them in reception in the accumulation case.

- Deferred and dismissed events. When an event that doesn't have any transition registered in the current state arrives, the default behavior is not to consider it, and therefore it is lost. In occasions can be interesting that this event is defferred to be processed later on. The deferred events allow to specify this behavior.

- Sequential and atomic processing of events. In dFSM, like in most implementations, when the system is executing an answer to an event (transition), new events are not processed until a stable in a state is reached. This non-preemptive nature is necessary to be taken in consideration in systems with very strict real time requirements. In this case, the most restrictive part must be implemented as perfectly identified automata, and would be the operating system who is responsible to guarantee the real time responsiveness.

- Concurrence. The Harel and-states have been implemented in a new way in the dFSM project. The programmer defines different automatas which will be executed concurrently with the communication mechanism. This approach has a certain parallelism with the distributed systems as CORBA, where the objects have its own execution thread (named Actors) and they exchange information through public interfaces, which they are subscribed previously.

- Object based. The dFSM internally and in the interface offered to the programmer is object oriented. C++ has been used as programming language.

- Actions. The final programmer can assign actions, code to execute, to states or to transitions, therefore, both Mealy and Moore automatas are supported. In respect to states, an action can be defined entering and another leaving it. This duality in the actions associated to the states is similar to the paper of an object's constructor and destructor. This means that each transition supposes the orderly execution of exit actions leaving the hierarchical structure, hereinafter the action associated to the transition and finally the enter actions of destination state ancestors. Execution engine. An engine has been developed to control the evolution of each automata, waits for incoming registered events. From a structural point of view (Figure 2), each automata has two threads, one dedicated to receive events and another to control the state evolution, transitions and their actions. Both threads exchange new events throgh a common object which is responsible of holding new events and keep mutual exclusion.

- Internal data structure. It is necessary to distinguish among the one that is shared by all the automatas and the one that is own by each one. In the first case it has been opted to locate it in shared memory. Mainly it is a table with registered events to every automata. This information is used in each occasion that you proceeds to send an event and you need to obtain the registered automatas. The connection is settled down by means of sockets UNIX. In the second case, each automaton has a list with the different states and transitions that it will be used to determine at all times which is the following state in function of the incoming events.

Finally, the programmer has an interface to code the automatas. It is contained in some shared libraries, to which the application will be linked. The LGPL (GNU General Lesser Public License) license has been selected to allow dFSM be used in proprietary applications. The administration is carried out in sourceforge.net portal, this fact that supposes to have an environment with typical services: CVS, mailing lists, forums, bugtraq,.... You can access from http://www.sourceforge.net/projects/dfsm or http://dfsm.sourceforge.net.

The project dFSM was born as a result of the support carried out by EUSS [7] university school and the firm LAIGU [8] to CIRSA INTERACTIVE [9] Corporation. Its main goal was to develop a framework to design embedded systema with the distributed automata paradigm. The project was defined and designed by the authors of this paper, and at the moment it is maintained outside of the environment of the commercial relationship.

4 High Availability

A majority of embedded systems need to operate in a high availability manner. In some cases this requirement is imperative, in others only an improvement to quality of service. As a framework that targets embedded systems HA is one of the project goals. To satisfy this requirement the following HA infrastructure has been developed. The dFSM project doesn't use redundancy, so the purpose in this case is to have mechanisms that allow the system to reestablish the service in case of faults.

Figure 3 shows a block diagram with our high availability infrastructure. At top level the different automatas are represented. Hereinafter the new element, the Supervisor, is shown; it is an agent that launches all automatas, supervises its behavior and reacts to malfunction if necessary. In turn the supervisor is monitored by the system watchdog[1]

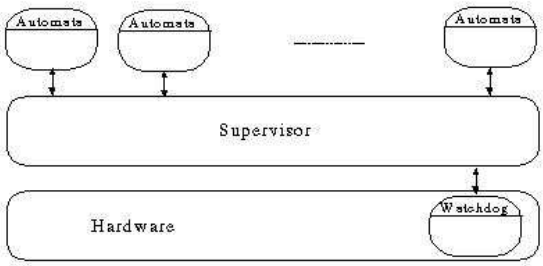

FIGURE 3: *High Availability System*

The implemented structure gives high availability based on three layers. Supports faults in the automatas through the supervisor, in the same supervisor (by means of the watchdog) and in the own operating system if a hardware watchdog is present. For supervision policy, in an initial version is supported the sequential launching of automatas (by means a set of dependence rules), monitoring the periodicity of some events generated by some automata and polling its heath state. In the future more complex policies will be supported. Two levels of behavior are distinguished when reacting to malfunction. In the first place a message can be sent to automata in order to trigger a recovery phase. If this this action does not bring the automata to a sane state, the supervisor can kill the process that contains the automata and restart it again. This process takes into account the dependency rules of the failing automata.

5 Sample Application

To show the usage of dFSM project, here is presented a minimal implementation of a robot that follows a black line on the floor. The robot has three reflective photosensors in the front part. By means of the information given by the sensors it decides to act individually on the movement of each main wheel, which can rotate forward, backward or to be stopped. The line follower robot has a rear free wheel.

The application that controls the movement has been divided into four automatas: `Wheel_right`, `Wheel_left`, `Control` and `Sensors`. Every automata has the states shown in Table 1. The automata communicates using the dFSM events communication mechanism already presented. In the Table 2 the exchanged events are presented, stressing who generates them and what automatas will be registered.

Each automata is modelled as a state diagram with its states, the transitions and the events. In the 4 the sensors state diagram is shown.

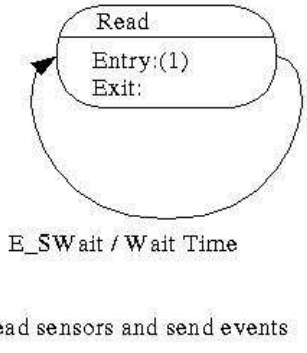

FIGURE 4: *Sensors automate state diagram*

[1]Timer that should be cleared periodically, otherwise causes a system reboot. It can be supported well by the hardware watchdog available in most embedded platforms, or by the software watchdog part of the LINUX Kernel.

Automata	State	Description
Wheel_right	Stop	Wheel stopped
	Move	Parent state
	Forward	Wheel moving forward
	Backward	Wheel moving backward
Wheel_left	Stop	Wheel stopped
	Move	Parent state
	Forward	Wheel moving forward
	Backward	Wheel moving backward
Control	Stop	Robot stopped
	Move	Parent state
	Forward	Robot moving forward
	Backward	Robot moving backward
	Right	Robot turning right
	Left	Robot turning left
Sensors	Read	Read sensors
	Wait	Waiting for next lecture

Table 1: Automatas and states

Event	Generator	Registered	Description
E_SRight1	Sensors	Control	Right sensor line
E_SCenter1	Sensors	Control	Center sensor line
E_SLeft1	Sensors	Control	Left sensor line
E_Stop		Control	Stop moving
E_Continue		Control	Continuae moving
E_SWait	Sensors	Sensors	Wait for a lecture
E_StopRight	Control	Right_Wheel	Stop right wheel
E_ForwardRight	Control	Right_Wheel	Right wheel forward
E_BackwardRight	Control	Right_Wheel	Left wheel backward
E_StopLeft	Control	Left_Wheel	Stop left wheel
E_ForwardLeft	Control	Left_Wheel	Left wheel forward
E_BackwardLeft	Control	Left_Wheel	Left wheel backward

Table 2: Events

In this case, the automata must poll the sensors and generate events if changes are detected. This task is carried out in the action associated at state entry. The event `E_SWait` is generated, the only listener to this event is this automata so a transition is triggered. The action associated to the transition is executed, in this case a time wait. The state diagram of one wheel is represented in Figure 5.

Highlights the parent state `Move` that contains the states `Forward` and `Backward`. This hierarchy allows to simplify the specification of the the transition that goes to the state `Stop`, because it is inherited by the two son states. Finally in the Figure 6 the state diagram of `Control` automata is shown. It is defined as a hierarchy with three movement states. Also in this case the benefits of transition inheritance are obvious. Pay attention to transition associated to `E_Continue` event from `Wait` state to `Movement` state, that is to say, the final son state is not specified, but an H bundled into a circle indicating that the final state must be the state of `S_Movement` before exiting or the default state if `S_Movement` has never been entered.

Finally the incomplete code is presented. Only pretends to show the state and transition definition. In the first place the states are declared like structures (they are objects that inherit from basic structures defined in the library dFSM). Note that an entry and exit actions can be defined. Hereinafter the transitions are declared with the same mechanism and also with an associate action. Finally in the main function the automata that contains the au-

tomata, states and transitions are defined and added to the automata object. Finally the execution begins. For each automata a similar code is need, so at the end 4 executables will be created.

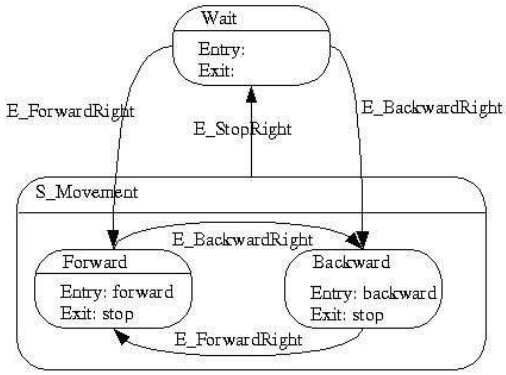

FIGURE 5: *State Diagram of right wheel*

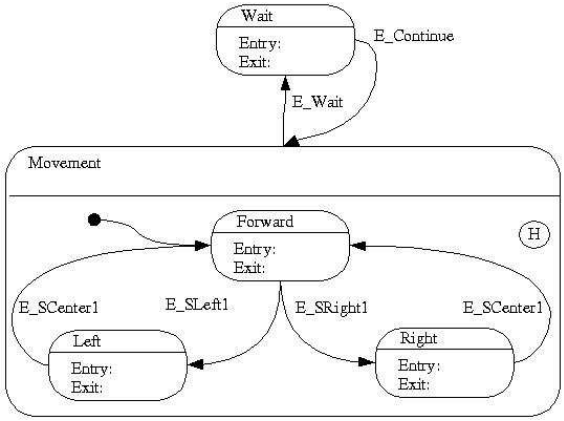

FIGURE 6: *Control state diagram*

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

struct tState_Forward :public tState {
    void entry_func(){ /* ... code */}
    void exit_func(){ /* ... code */}
};
/*...*/
struct tTransition_Movement_Forward:public
    tTransition {
    void function(){ /* ... code */}
};
/*...*/
int main (int argc, char* argv[])
{
    tFSM *fsm; tState *s0, *s1; tTransition *t0;
    fsm  = new fsm("Nombre automata",0);
    s0 = new tState_Movement();
    s0->set_id(0); // State identification
    fsm->add_state(s0); // Add state to automata
    s1 = new tState_Forward();
    s1->set_id(1);
    s1->set_parent(s0); // Set hierarchy
    fsm->add_state(s1);
    /*...*/
    t0=new  tTransition_Movement_Forward ();
    t0->set_source(1); // Source state
    t0->set_target(2); // Target state,
                       // State Wait = 2
    t0->set_event(0);  // Event
    fsm->add_transition(t0); // Add transition
    /*...*/
    fsm->run(); // Run automata
}
```

6 Related work

As already mentioned, Harel[2] revolutionized the automata theory with the inclusion of hierarchy and concurrence. This whole base has been materialized in Statemate[10], an application that allows the analysis, design, and the later automatic generation of code for reactive systems. Is a software with wide benefits, is a reference in their sector but completely proprietary.

The automatas are also part of the modeling language UML. They are used to represent behavioral models. In this case, the goal of different tools that work with UML is more wide and generic than the design of reactive systems. Many tools support a final step to generate code of the modelled system and they could be used like base to design this type of systems. Good examples are Rhapsody of I-Logic, Rational of Rational Software Corp. and ObjectGeode of Telelogic. In all these cases are proprietary products. In the free software field, one or the alternatives to modellate with UML it is Umbrello, package with limited characteristics regarding the previous ones, but it can be enough for many applications. The problem of this implementation resides in the fact that they have been conceived with a wider optics as for the system type. This is manifested in a lack of efficiency in the generated code for the concrete case of reactive systems.

Samek[5] presents a different approach. In this case it is not the case of a package, he has developed an infrastructure (denominated Quantum Framework), a library that allows a systematic programming of distributed automatas. In this sense, the approach is similar to the dFSM one, but with marked structural differences in the code gives the states and the automatas.

In the open source field, the most outstanding approach it is the project SMC (State Machine Controller)[11] that outlines it as a precompiler in which

by means of a pseudo-language translates the specification of an automata in source code C++. In general it supports most of the functionalities but with an approach different to the hierarchy, which is based on comparing a sub-diagram of states to a subroutine. Neither doesn't carry out an explicit treatment of the concurrence.

Robotics and control world has two important free software projects. On one hand MatPLC[12] which has created an infrastructure to be able to design a system like a group of modules, each one of them implemented as a PLC (Programmable Logic Controller, device gives utilized control thoroughly in the field of the industrial automation that is characterized by the simplicity in its programming). On the other hand, the project OROCOS[13] it approaches the design of a control system but in this case with a platform based on components inspired by CORBA.

The project dFSM, presented in this paper, depart from the automatas of exposed works and approaches the topic of the concurrence in a singular form up to now. The frequent pattern that implements an application with a group of processes that collaborate between them can take advantage of the dFSM project putting an automate inside each process.

7 Conclusions and future directions

In this article the project dFSM has been presented like programming framework that allows to simplify and to systematize the design and implementation of systems based on distributed automatas. The records that have motivated the work and the dFSM solution have been presented. Finally an example that shows the simplicity of code and a comparative with the existent work have been presented.

At the moment the library its enough stable to be considered as an alternative in many projects of embedded systems. The work continues improving the stability and features. In the first place a debugger system has been developed. Allowing the monitorage of each automata evolution and observe the system response in front of manually inserted events. In second place, a graphical tool is being developed in order to carry out the design phase in a graphic environment and generate a skeleton of the different automates once the design is finished. Finally, when the stabilization reaches a satisfactory level, the development will focus on reduce dependencies within STL library and portability.

References

[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 Sarrià (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] CFSM project, www.sourceforge.org/projects/cfsm

[12] MatPLC, mat.sourceforge.net

[13] OROCOS Project, www.orocos.org