STUDIA UNIV. BABEŞ-BOLYAI, INFORMATICA, VOLUME II, NUMBER 1, 1997

SPECIFICATION OF ACTIVE OBJECTS BEHAVIOUR USING STATECHARTS

VASILE MARIAN SCUTURICI IULIAN OBER
MIHAELA SCUTURICI DAN MIRCEA SUCIU

Abstract. Object oriented concurrent programming is a methodology that seems to satisfy nowadays requirements for complex application development on multiprocessor architectures. The fundamental abstractions used in this methodology are active objects and protocols for passing messages between them. The field of object oriented concurrent applications analysis and design and also the development of CASE (Computer Aided Software Engineering) tools that assists designers of this kind of applications are current research issues. In this context, statecharts seem to be one of the most appropriate ways of modelling the behaviour of active objects. Therefore, we developed an implementation model of statecharts in an object oriented programming language (C++). This model simplifies the implementation of active objects and it can be integrated into an automatic code generation tool from a specification or design model.

1. Introduction

Concurrent object oriented programming is a promising methodology. It allows us to describe problems using collections of entities (called objects) that embed some properties, perform computations and concurrently interact through a unified communication protocol (called message passing). This methodology represents an excellent basis for program decomposition and efficient execution of them on multi-processor architectures.

Concurrent object oriented programming is applied in many fields. There are applications developed in the following fields:

- distributed operating systems
- distributed artificial intelligence
- distributed simulation
- distributed data bases
- real-time systems and the control of distributed processes

Significant results were also obtained in computer-aided text and music processing. Concurrent object oriented programming has an important impact on multi-processor architectures design.

Received by the editors: November 25, 1997.

1991 Mathematics Subject Classification. 68Q10, 68Q90, 68Q05.

1991 CR Categories and Descriptors. D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Tools and Techniques - *Computer-aided software engineering (CASE), Software libraries.*

In section one we present the main concepts related to concurrent object oriented programming and we define the active object concept.

Section two contains the presentation of statecharts and the advantages for describing object behaviour (including the active ones) and for facilitating source code generation process.

Section three presents an implementation model used for active object description using statecharts. Is also presented a way for using this model in C++.

The remarks in section four establish the advantages and drawbacks of modelling active object behaviour using statecharts and suggest future research directions in this field.

2. Active objects

Concurrent object oriented programming is based on object oriented programming methodology which is known at this moment as a top methodology for developing applications with and for reuse. This methodology is conceptually simple and wide applicable. It is based on two fundamental concepts:

- *objects* – that identify knowledge (represented by information and services), and
- an unified *protocol* for activating and communicating between objects, called message passing

Objects represent entities and concepts that form the problem domain. Applications developed in an object-oriented way have no global algorithm, but rather they are composed of a set of objects. Objects cooperate by sending each other requests and exchanging information. They are animated like in a simulation. Other fundamental concepts, intimately related to object oriented programming, are *abstraction* (the *class* concept) and *inheritance* (the *subclass* concept) which allow specification, classification and reuse of object descriptions.

The first programming language based on these concepts, Simula67 appeared at the end of 60's. Initially, concepts were descending from a simulation language in a general programming environment. Programming is regarded as a simulation of the real world. The concepts evolved since then but even in its rough form from Simula67, object oriented programming was regarded as a significant improvement in software development and a rise in quality, robustness, reusability, extensibility.

Due to the sequential nature of conventional programs and processors, the autonomous activities of each object – that, naturally, led to simultaneous activities – has not been treated by object oriented programming systems in the native parallel way. Therefore, in most cases the programming languages were and continue to be sequential. This means that, at a moment in time, only one object is active and this activity is passed from an object to another by synchronous messages.

On the other side, the ability to express the potential parallelism of activities through concurrent programs became a research subject for the next generation of systems. These are only some aspects that suggest the importance of this tendency in programming:

- The development of supercomputers that allow parallel execution of multiple threads, increasing programs' efficiency

- The distribution of information through a computer network
- The existence of multiple interaction inside some large interactive programs (like processes control, where the inputs and the control have to be multiple and distributed)
- The natural way of expressing certain applications using concurrency and activity co-ordination mechanisms

Considering the above assertions, the idea of building a programming language that can integrate the object oriented programming mechanisms with concurrency mechanisms is very attractive. To achieve an optimal integration of these mechanisms is very useful to identify objects as activity units and to associate synchronising code at the message passing level. These *active objects* are often called *actors*. The result of this unification is the integration of all the object oriented programming and concurrency concepts and it allows the programmer not to be explicitly involved in establishing the synchronisation discipline. It also maintains the modularity and simplicity that is typical for the object-oriented programming, and respects object's embedding and autonomy.

This integration represents a natural generalisation of object oriented programming, giving a greater autonomy to the objects. In [2] J. Brriot says that object oriented programming is a technological restriction of the more general concept of concurrent object oriented programming.

In concurrent object oriented programming the concurrency can be modelled at two levels:

- *Inter-object concurrency* – that means parallel execution of autonomous activities by more than one object.
- *Intra-object concurrency* – that takes the concurrency at the object level by giving the possibility of treating concurrently more messages on the same object.

Message passing is the co-ordination and interaction mechanism between objects. All interaction types, from information extracting to requiring an object to perform some computation, are covered by the message passing concept. A message passed to an object determines the activation of a method (an operation associated with a message pattern) of the receiving object with the arguments contained in message.

Due to the potential competitions between source object activities and those of the receiving object, it is useful to store the messages in a waiting queue for managing communication speed differences between the receiver and the source and also for releasing the source object when a response is not necessary.

2.1. Statecharts

Statecharts are used for specifying objects in designing complex systems. Statecharts constitute a visual formalism for describing states and transitions in a modular fashion, enabling clustering, orthogonality and refinement. In statecharts we use the elementary notions of state and event. A state expresses a certain condition that an object satisfies or a conjuncture that it is in. At different moments in time an object can be in many states. An object can receive (observe) events regardless of the state it is in, but it usually responds to them only when it is in an appropriate state. The response to an event is drawn on the statechart by a transition that links a state in which the object responds to that event and the state reached by the object as the result of responding to

the event. If the occurrence of the event does not change the state of the object, it is an autotransition (the final state is the same with the initial one).

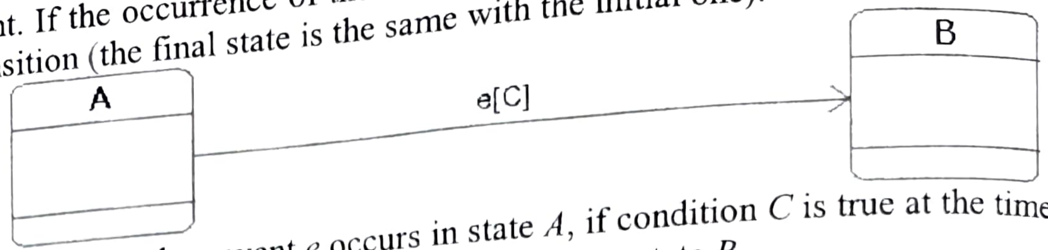

Figure 1: when event *e* occurs in state *A*, if condition *C* is true at the time, the system transfers to state *B*

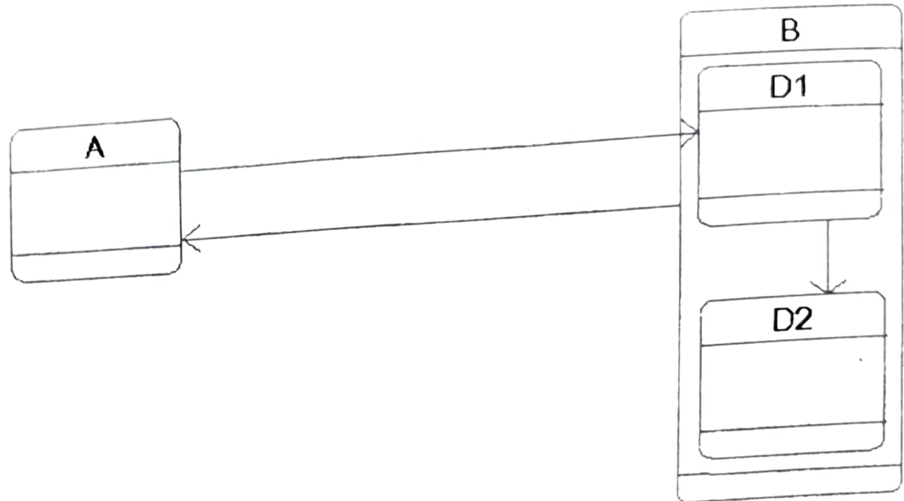

Figure 2: Nested states in a statechart

In this way we can build a graph with nodes representing states and labeled arrows representing transitions. This graph is a transition diagram between states (a finite state machine). For avoiding the growth of the number of transitions, we can use nested states.

For avoiding the exponential growth of the number of object states generally due to the fact that certain objects have independent subparts that function each as a state machine, we can use orthogonal state machines. An example is given in a later section. For details see: [1], [6], [7], [8], [9], [10].

The problems that appear are related to statechart modelling in object oriented programming languages. In this article, we will present a possibility for modelling active objects with statecharts in a language that does not support them natively, (C++).

2.2. *Using statecharts for active object description*

Reactive objects in an application can be described using statecharts. However, in object oriented programming languages (like C++, Smalltalk, etc.), the notions of state and event, as we understand them here, do not exist. We established that in order to substitute these lacks we had to develop an implementation model like an extension of the C++ programming language.

In order to have a sufficiently general model, we supposed that all objects are potentially active objects. Similarly, all existent objects may receive the events, but only some of them will respond to them, generally by calling a method.

An object created within our model will contain the attributes and the methods defined in its class and also an extension that will allow it to use statecharts.

The extension will be created automatically. The new object has all the features of the old object, further allowing some additional behaviour.

Figure 3

We will present here the extension of an object (extension whose implementation is hidden, but is useful for the understanding of the model).

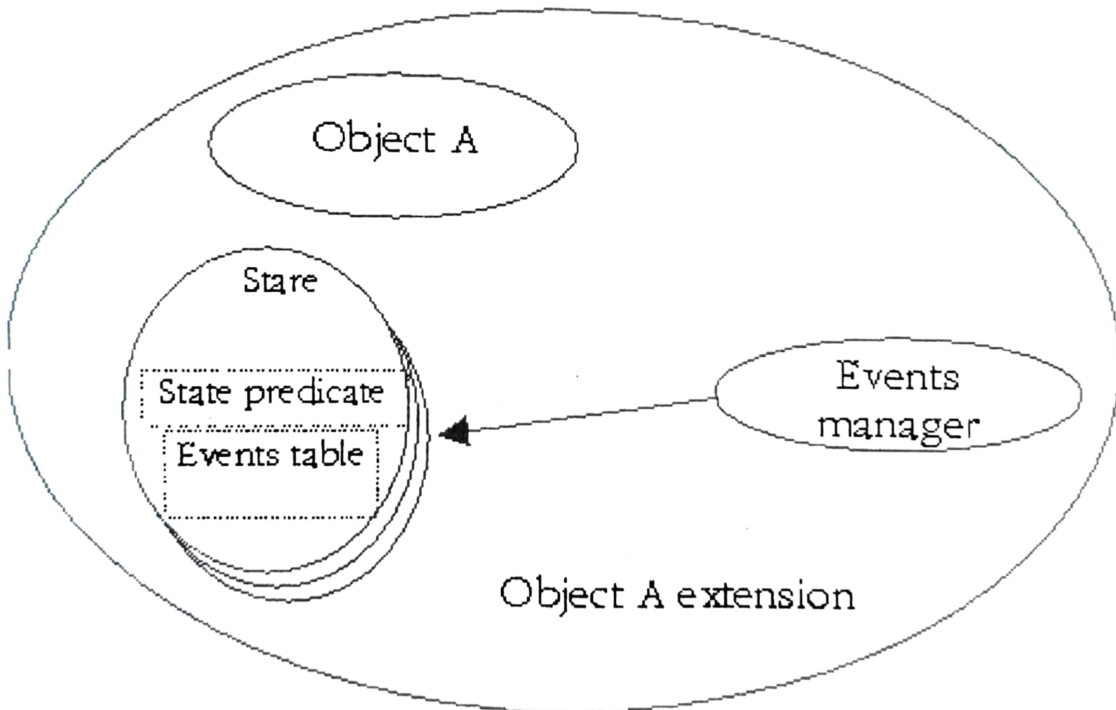

Figure 4

The new object will contain:

- *The initial object*: all its attributes and methods.
- A number of *state objects*. These are data records containing information related to the states defined in the statechart. They contain the state *invariant* which describes the condition or conjuncture modelled by the state. This invariant is used in order to check whether the object is in a certain state or not (the object is in the states for which the invariant holds true). They contain also an associative table. Each element in this table will be a tuple *(event, method,*

precondition, postaction) and it means that when the event occurs for an object that is in that state, the method will be called (just in case that *precondition* is true). After the method execution the *postaction* will be executed.

- *Events manager*. The events manager is an interface of the new object for the application event flow.

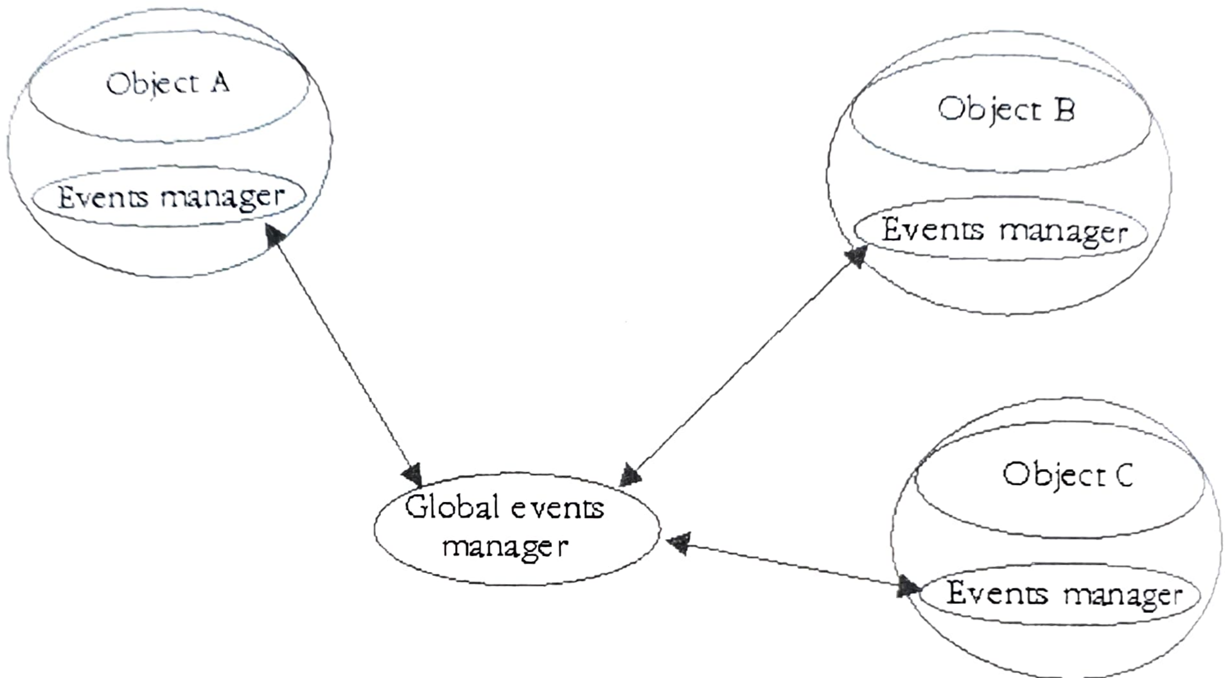

Figure 5

The application will contain a global event manager. That is an object that manages all the events that circulate between the application objects. It allows communication between applications by events, in a way that will be presented bellow.

Thus, an object can send an event to any other object using his event manager. This event will be received by all objects that have the event managers connected to the global event manager (if a destination is not specified).

The working mechanism will be the following:

- An object sends an event through its manager to the global event manager
- The global event manager sends the event to each object event manager connected to it
- Each event manager finds out which are the current states of the corresponding object (there can be more valid states at a moment if we have an orthogonal diagram).
- Inside that state it searches all the entries in the events table, entries that have the same name as the received event (there can be more than one entry because of the preconditions). If such an entry does not exists that means that the object does not know or want to respond to that event. If the entry exists and the precondition is satisfied, the corresponding method is called.

2.2.1. States organization

Inside an object the states (as objects) are organized as a tree, allowing the modelling of nesting and orthogonality and, also, a better treating of the events between them.

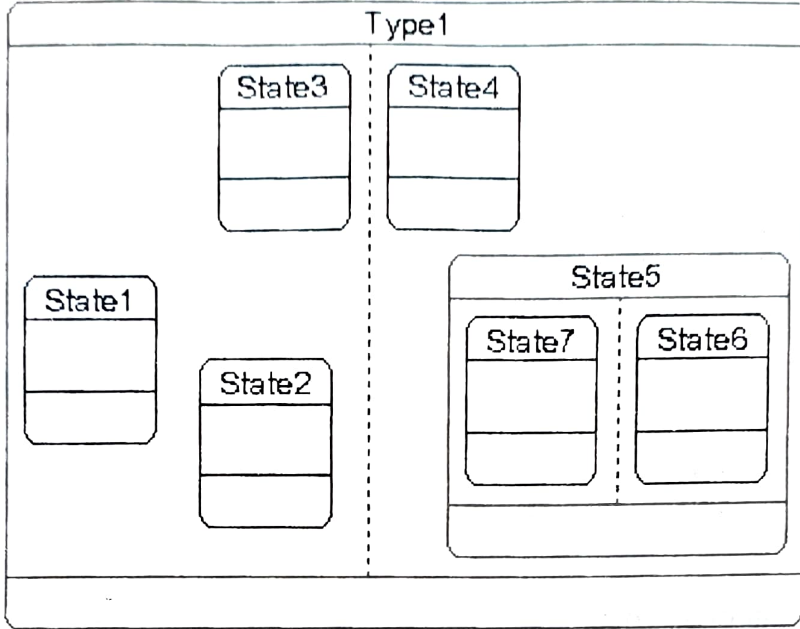

Figure 6

For this statechart, the representation of the states inside the object will be the following:

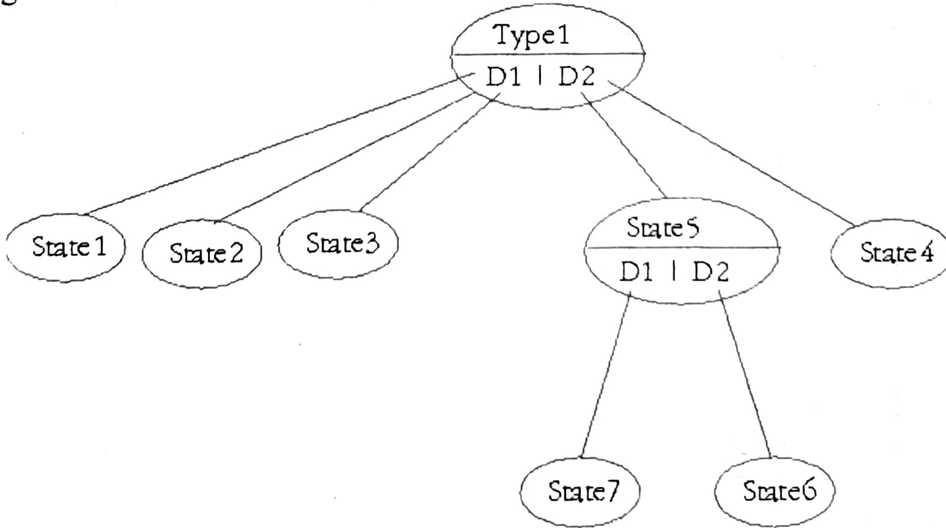

Figure 7

When an event appears in the root, it is sent to all the subtrees. Inside a node, an event is sent to all the composite states of the node.

2.3. The C++ implementation of the presented model

We will see now the way that we implemented the presented model in C++. We will take the following example that describes an active object class. An object of this

class can be created, can be started for continuously executing an activity (a step) after that it will be stopped.

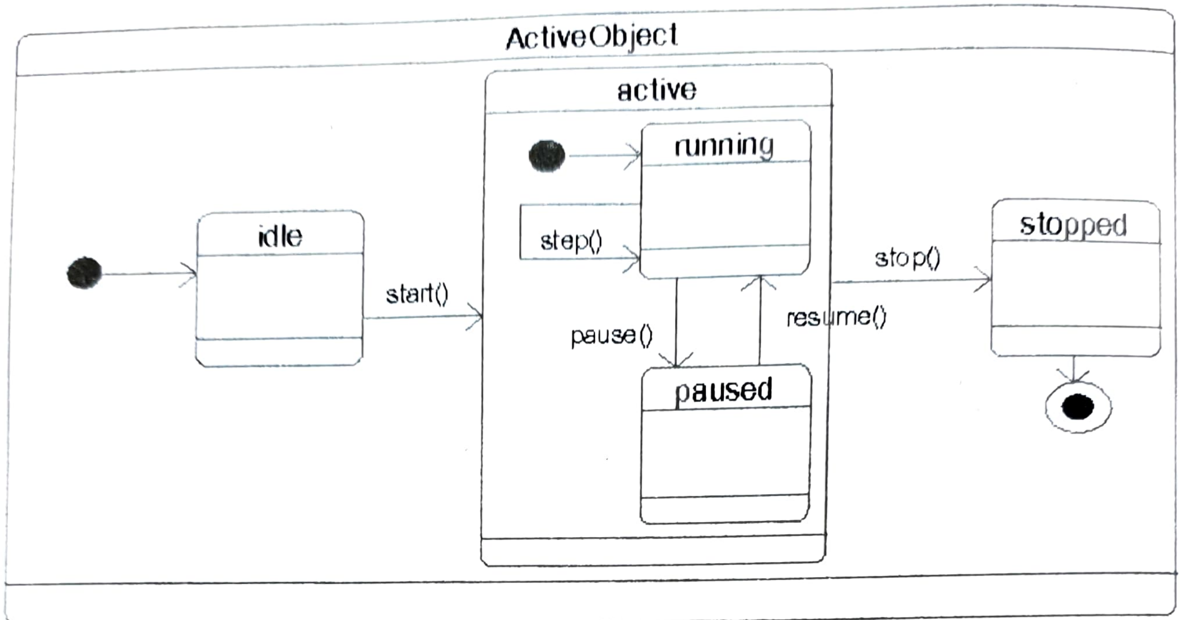

Figure 8

At the moment of creation, the object is in the *idle* state. When it receives the *start* event, the object passes in the *running* state. Here it may receive the *step* event, remaining in the same state and executing the *Step()* method. When it receives the *pause* event, the object will pass from *running* into *paused* from where it may return at the receiving of the *resume* event. The activity of the object ends when it receives the *stop* event when it is in the *active* state (that means one of the *running* or *paused* states)

The implementation in Visual C++ for this class using our framework:

```
class ActiveObject : public FSM
{
	CString m_state;
	CWinThread* m_runningThread;
public:
	ActiveObject();
	~ActiveObject();
	virtual void Step(); // method called at the receiving of step
	void StartObject(); // starts an execution thread inside the object which sends
				//all the time the step event to the current object
	void StopObject();
protected:
	// the associated statechart...
	DECLARE_FSM
		STATE(idle, ROOT, 0, m_state == "idle")
		STATE(active, ROOT, 0, (m_state == "running")||( m_state == "paused"))
		STATE(stopped, ROOT, 0, m_state == "stopped")
		STATE(running, active, 0, m_state == "running")
		STATE(paused, active, 0, m_state == "paused")
		TRANSITION(start, idle, active, startObject, TRUE, m_state = "running")
		TRANSITION(stop, active, stopped, StopObject, TRUE, m_state = "stopped")
```

```
        TRANSITION(pause, running, paused, NULL, TRUE, m_state = "paused")
        TRANSITION(resume, paused, running, NULL, TRUE, m_state = "running")
        TRANSITION(step, running, running, Step, TRUE, NULL)
    END_DECLARE_FSM
};
```

If we want the objects of a class to be able to work as state machines, it is necessary to derive the class (directly or indirectly) from our FSM class, that is a part of our framework.

The description of the statechart is made in the following section of the class definition:

```
    DECLARE_FSM
        // statechart section
    END_DECLARE_FSM
```

The macros used are:

```
    STATE(newState, parentState, stateDiagram, stateDefinition)
```

where:

- *newState*: is a new state that will be added to the class's statechart
- *parentState*: the parent state in which the newState is introduced; it may be ROOT (in which case it does not have a parent)
- *stateDiagram*: the number of the orthogonal state machine of the parent state where it will be added; it will be 0 if we do not have orthogonal states;
- *stateDefinition*: a Boolean expression that will be evaluated each time the event manager wants to determine whether the object is in a certain state or not; this is the state predicate;

```
    TRANSITION(event, currentState, newState, method, precondition, postaction)
```

where:

- *event*: the name of the event that triggers the transition
- *currentState*: the source state of the transition
- *newState*: the state in which the object will have to be after the receiving of the event
- *method*: the name of the method that will be called when the *event* occurs; it can be NULL, in which case nothing will be executed;
- *precondition*: a Boolean expression that will be evaluated first of all; only if this is TRUE, the object will begin the execution of *method*; if it is FALSE the transition will not be executed;
- *postaction*: a sequence of instructions that will be executed after the execution of *method*

2.3.1. The Inheritance of the Statecharts

With simple inheritance, when *B* inherits from *A*, if *A* has defined a statechart and *B* defines other states, these will be taken as a supplement of those that are in *A* (which will be found in *B* too). With multiple inheritance, the class that inherits from other classes inherits the statecharts too. The statecharts of the parent classes will be found as

orthogonal diagrams of the heir class. The statechart of the new class will complete the diagram of the first class in the parents chain.

2.3.2. Concurrency support

In the FSM class there are many methods that can be used for modelling concurrency. One method that verify if the object is in the state named ***stateName*** is:

```
BOOL AtState(CString stateName);
```

We remember that an object may be in more than one state if it has orthogonal machines.

The method that is called when receiving an event is:

```
void ReceiveEvent(CString eventName);
```

Normally this is used by the global event manager.

The method that sends an event using the principle "who has ears to hear let it hear":

```
void SendEvent(CString eventName)
```

When this method is called, the current thread is locked until all the objects that know how to respond to this event respond to it.

Another method that implement the same mechanism as the one above, with the difference that an asynchronous event is sent is:

```
void ASendEvent(CString eventName)
```

This means that it will not wait until the sent event finishes the execution, it will continue concurrently:

The method that causes the stop of the current thread until *pObject* arrives in the *stateName*: state:

```
virtual void WaitUntilObjectAtState(FSM* pObject, CString stateName);
```

There are also methods for modifying the statechart at runtime. One can: add states, add transitions, modify them.

2.3.3. Support for working in a distributed environment

For allowing the interaction between applications that are running on many computers, we can use the extension of the implementation of statecharts. How will it work?

The global event managers are connected with each other and the events circulating inside an application will arrive also in the applications that have the global manager connected to the former one.

The methods that allow connecting the global events managers between applications are:

```
BOOL InitServer(int nPort);
BOOL ConnectToServer(int nPort, CString serverName);
```

- *nPort* is the TCP port number on which the applications will communicate

- *serverName* is the server name to which the connection is attempted (internet name or address).

For instance, if we want to use synchronisation between an application that has an AVIPlayer object and another application that has a Timer object (these classes are descendants of FSM), we will do it this way:

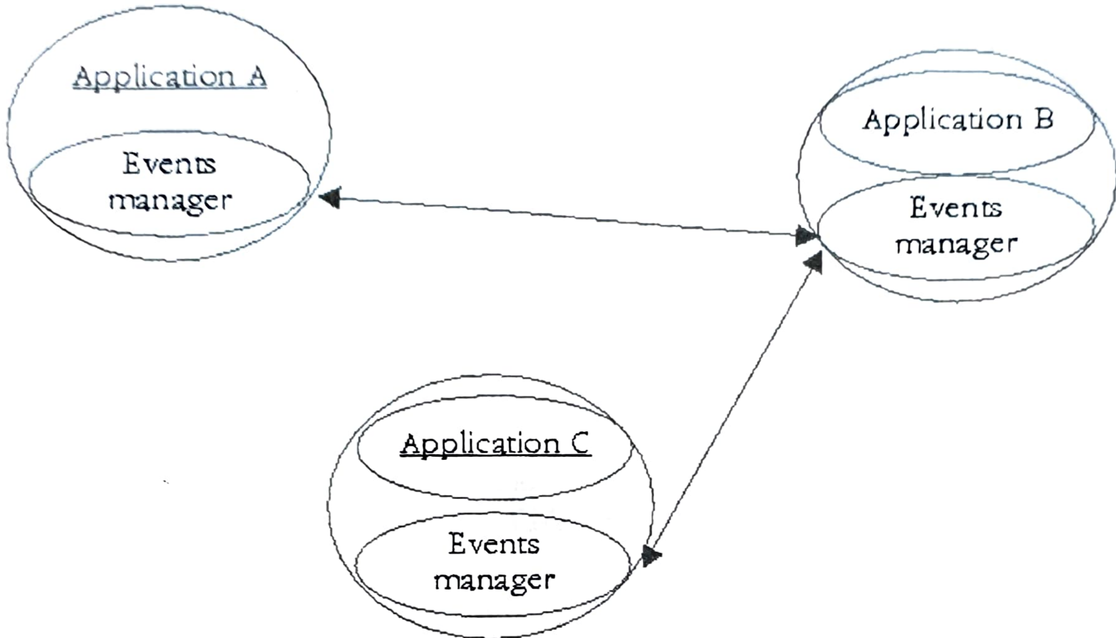

Figure 9

In the application that contains CAVIPlayer we will introduce the following code:

```
CAVIPlayer* pPlayer = new CAVIPlayer("clock.avi");
pPlayer->InitServer(711);
```

If we have already created the object that will display a clock on the monitor and when receiving a FSM_TIMER event, it will advance one position, and we have initialised the server on the port 711 (we work on the machine with the IP address *apollo2.cs.ubbcluj.ro*), inside the application whose global manager will be connected to the previous one we will put:

```
Timer* pTimer = new Timer(1000);
pTimer->ConnectToServer(711, "apollo2.cs.ubbcluj.ro");
pTimer->StartObject();
```

It has been created a Timer object which send the FSM_TIMER message at one second; connected to the application on the apollo2.cs.ubbcluj.ro will start to display the movement of the clock arrows which advance at each second, as a response to the received event.

3. Conclusions

The most important benefit of this model is that it allows using statecharts into object oriented programming languages.

The statecharts of an object are initialised with the class pattern, after that they can change their configuration during the execution; in other words, the structure of statecharts can be dynamically modified for each object; this is not specified in object oriented analysis and design methods.

Because the mechanism of working with macros for the statechart description is rather difficult (but viable!), it is useful to use a CASE tool for automatic code generation for the statecharts; this should not be manually modified, or completed.

Working with this implementation of statecharts is useful in case we can describe a class using a statechart. If all the classes of an application can be described this way, then we can speak about 100% generated code (not only 100% ***correct*** but also 100% ***complete***).

Some problems that appear are related to statecharts inheritance. Here we have just an approach, but it does not solve all the problems.

REFERENCES

[1] David Harel, Statecharts: *A Visual Formalism for Complex Systems*, Science of Computer Programming, North-Holland, 1986

[2] Jean-Pierre Briot. *Object-Oriented Concurrent Programming: Introducing a New Programming Methodology*,. Proceeding of the 7th International Meeting of Young Computer Scientists (IMYCS'92), Topics in Computer Science Series, Gordon & Breach, 1993

[3] Michael Papathomas, *Behaviour Compatibility and Specification for Active Objects*, Pattern Languages of Programming conference in Monticello, Illinois, September 6-8, 1995

[4] Lavender R. G., Schmidt D. C., *Active Object, an Object Behavioral Pattern for Concurrent Programing*, Object Frameworks, D.Tsichritzis eds. Centre Universitaire d'Informatique, University of Geneva, 1992

[5] Michael Papathomas, G. S. Blair and G. Coulson, *A model for Active Object Coordination and its use for Distributed Multimedia Applications*, ECOOP '94 Workshop on Coordination Models and Languages for Parallelism and Distribution, Bologna, Italy, July 1994

[6] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall, 1991

[7] James Rumbaugh, *Modeling & Design, Journal of Object Oriented Programming*, 1993-1995

[8] Sally Shlaer & Stephen Mellor, *A Deeper Look*, Journal of Object Oriented Programming, 1993-1995

[9] Sally Shlaer & Stephen Mellor, *Object Lifecycles. Modeling the World in States*, Prentice Hall, 1992

[10] Steve Cook, *Object Oriented Design with Syntropy*, Prentice Hall, 1994

Babeş-Bolyai University, Faculty of Mathematics and Informatics, RO 3400 Cluj-Napoca, str. Kogalniceanu 1, România.

E-mail address: {scuty, iulian, mihaelas, tzutzu}@cs.ubbcluj.ro