Cellular Objects for Cellular Computers

Emmanuel D. Adamides

Institut d'Automatique
ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE
CH-1015, Lausanne, Switzerland
e-mail: adamides@elia.epfl.ch

1. INTRODUCTION

The term cellular computer can be associated with a number of computational engines that range from chip-level architectures to massively parallel systems of the size of Connection Machine, and whose common characteristics include the relative large number of simple processing units with limited local memory, as well as, the simple (mostly synchronous/broadcasting) communication protocols. From a different point of view, these systems can be viewed as extensions of the simple cellular automaton model of computation. The extensions include properties, such as, non-homogeneity of the cells/processors, non-local addressed communications, etc. [Hillis85].

Cellular computers with conventional software environments have been used in many applications in the areas of symbolic processing (sorting, unification, etc.), scientific simulation, graphics, even financial simulation and forecasting. However, for some large-scale process-based business and industrial applications, these environments, and their corresponding application domain modeling techniques, seem to be very limiting. On the other hand, the general methods and tools used to describe these applications have always been quite conservative. For instance, although an enterprise is a highly parallel system of interacting processes, most enterprise-wide information models are far from parallel computation models. This contributes a lot to the very weak association between powerful parallel computing and knowledge processing enterprise-wide applications.

Cellular automata have inherent object orientation characteristics but they are too fine-grain and too rigid to serve as a distributed process-based modelling formalism and, consequently, as a software development paradigm. A higher level, well-structured and more flexible model is required to act as a basis for active data structures. To take advantage of the developments in object-orientation, we need to develop "light" active objects as an intermediate option between distributed intelligent agents (or active objects) of the type of *ACTORS* (or even coarser-grain) and N-dimensional n-bit CA which will be able to execute in relative simple processing elements communicating in a very simple way. In response to these needs we have developed *cellular objects* which are briefly presented in this paper.

2. CELLULAR OBJECTS

Since concurrency seems a natural consequence of the concept of objects, there has been quite a large effort on the marriage of objects, distribution, concurrency and autonomous behavior, seen from various points of view and with different perspectives [Chin91]. The most well known model of concurrent active objects is that of *ACTORS* developed by Agha [Agha86].

Structurally, a *cellular object* encapsulates attribute values and links, as well as methods to manipulate them. Behaviorally, it is an *agent* capable of running a processes communicating with its environment *synchronously*. Its environment consists of agents (cells) with the same or different characteristics. Cellular objects are autonomous self-sufficient software units, residing on different processors, whose behavior *depends* on other objects, but it is *completely* under the control of the object itself The object, itself, decides which method to execute based on the state of its neighboring objects. This means that in addition to the private data structure of an object which is inaccessible, its methods are also directly inaccessible by the other objects.

Compared to *ACTORS* that use asynchronous buffered communication, cellular objects use (SCCS-like) broadcasting for communication. Only the states of objects are communicated. While *ACTORS* can be created dynamically at run time, cellular objects are defined statically at compile-time and they "live" until the end of the execution of the program. However, they may be connected or disconnected from the rest of the environment, at run-time, as dictated by the state of the environment/neighbor objects. This is done by manipulation of their connecting ports. In order to cope with real-time cooperation-intensive application requirements (full determinism), and at the same time be simple, instantiated cellular objects use the static notion of one-object/one-process/one-processor. This scheme remains constant during run-time.

Formally, a cellular object is defined by the 8-tuple

$$O = (o, Q_{st}, Q_m, \Sigma_{acc}, r, m, H, M)$$

where:
o is the name of the object,

Q_{st} is the set of states of the object as they are defined in the data structure,

Q_m is the set of states which are used for the selection of the communication ports and the methods to be executed (*mode selection states*),

Σ_{acc} is the set of acceptable input events that are limited to the parallel composition with full synchronisation of the of the states of the neighbour objects,

r is the set of neighbours of the particular cell,

m is the method selection function

$$m = F(q_{st}, q_m, \sigma_{acc}, H)$$

where
H is the *neighbor vector* which supplies the information regarding which neighbor of each of its neighbours the specific object is, and finally,

M is the set of the methods that manipulate the object's states including the mode selection states. All methods are functions of the elements of Q_{st} and Σ_{acc}.

The instantiation of a number of cellular objects forms a *cellular object graph (COG)*. The COG is paced by a global clock as cellular objects always communicating with each other. They do so, even when they have nothing to say or even when change of state means "freeze at the same state". A cellular object, at each time step, senses the states of the (interesting) neighbours, reads its neighbor vector, chooses one of its methods that will execute, and finally changes its state according to this method. All methods manipulate, in some form or another, the current state of the object, as given by

$$\textit{current state} = \textit{data state } (q_{st}) + \textit{selection state } (q_m)$$

The global cellular object evolution function is the synchronous concurrent composition of the methods that are executed at every time step, and which maps a global system state of the graph at a time instant to a new global state of the graph at the next time instant. Apparently, the global object matrix states represent the states of the global application domain. Figure 1 shows the internal as well as the communications structure of two active cellular objects.

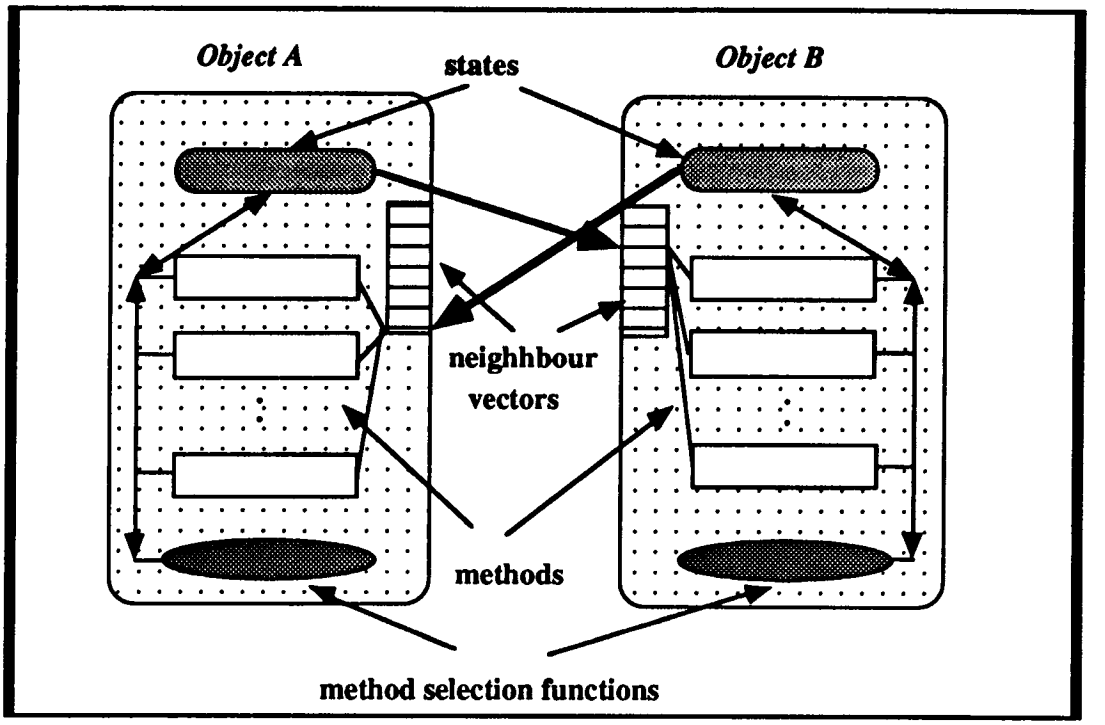

Figure 1 : Cellular Objects

Communication between the cellular objects themselves, as well as between the cellular objects and the external environment is *fully synchronous*, based on the so-called approach for real-time and reactive systems (e.g. see [Benveniste91]). In addition, cellular objects are *reactive* in a sense that their behaviour is simply their reaction to the information broadcasted by the other objects.

According to the characteristics of the synchronous approach to reactive real-time systems, the (state) output of an object is synchronous with its input, the execution of the invoked method is instantaneous and the communications are performed via broadcasting, i.e. the state of the object is broadcasted to the connected cellular objects. If it is not used by the connected objects this information is lost. There are no message queues as in the (asynchronous) *ACTORS* model.

Figure 2 shows in some detail the synchronous operation and communication of two cellular objects. Initially, the method selection function is executed with the current and the neighbor states (as it is seen through the neighbor vector) as parameters (1,2). Following, the selected method is executed and the result updates the object state (3). Finally, the object state is broadcasted to the neighbouring objects (4). It should be noted that these steps form an *atomic operation* and they cannot take place separately, i.e. once the operation starts, all steps have to be performed. Being reactions, they take no time in the sense that they take no time with respect to the external environment which remains invariant during the operation. In absolute terms, they take one time unit of the global clock. The period of the global clock should be long enough to allow for the object with the larger method to execute.

Cellular objects that have no methods (only data structure definition) are called *dummy cellular objects* . Their states are set and reset externally in the same way that static memories are set and reset. In addition there are *command* or *interrupt objects* which store only the name of the method which <u>must</u> be executed with the highest possible priority, no matter of the state of the rest of the cellular object graph, at the next time step. In other words, they provide the buffered channels through which interventions from the external word (the user). Auxiliary objects to implement counters of time instances are parametric in the number of instances, and a sufficient number of such objects has to be provided for each control architecture.

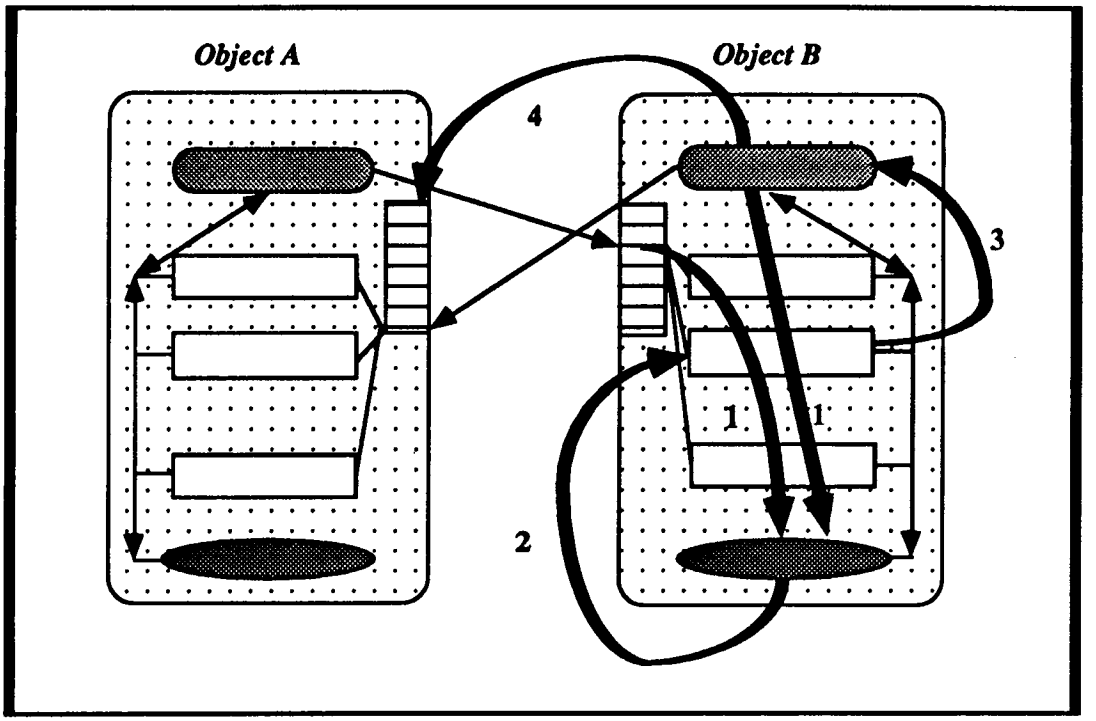

Figure 2: Operation (reaction) of Cellular Objects

3. THE USE OF CELLULAR OBJECTS

So far, cellular objects (CO) have been constructed only as discrete-event simulation models for two, apparently, diverse applications, namely factory-level automation [Adamides92a] and Computer Supported Cooperative Work (CSCW) [Adamides92b]. Both, however, within the framework of our effort towards fine-grain enterprise-wide process-based modeling with the final aim to use cellular computers as information processing machines for the non-hierarchical cluster-based enterprise of the future.

In the first application, for the first time, notions of massive parallelism have been introduced into the factory, as a large number of CO is used to perform, employing distributed knowledge-based techniques, the factory supervision task. The overall control task is decomposed into sub-tasks of cooperative, fully deterministic control demolishing all the unproductive decision and control hierarchies of the shop floor. All factory decision points are substituted by active cellular objects which perform the control task by cooperating with each other, as well as, by interacting with the physical environment.

Since in most modern flexible factories, machines and their related hardware are organized in various *islands of automation* integrated into larger units such as Flexible Manufacturing Systems (FMS), coordination should be provided at, at least, two levels: at the level of individual cell component, i.e. the machines and the material and tool transportation systems, as well as at the level of the shop-floor where the operation of the different cells and their interactions have to be coordinated.

Multi-level structures where cellular object graphs are coordinated by cellular objects which belong to higher level cellular object graphs can be used for the coordination of cellular shop-floors. By using a systolic-like information flow [Adamides92c] between the various control levels, coordination at the factory level can be achieved. Special cells, called *transporters*, transport state information to different level cellular graphs. Transporter objects contain special methods called *filters* that automatically "filter" the state of the object before it is broadcasted to a different level cells.

For the case of the CSCW, real-time distributed meeting scheduling has been chosen since it can be decomposed into a large number of fine- to medium-grain cooperative tasks. The structure of the cellular graph represents the structure of the application environment which, for the case of

large organizations, may be enormous. Hierarchical decomposition techniques based on CA hierarchical structures [Adamides92c] have been employed. Cellular objects represent the individual participants processes and schedule meetings by cooperation. Having the synchrony property, cellular objects overcome the real-time burden quite easily. Figure 3 shows, in more detail, the structure of a cellular object used in this application (Figure taken from [Adamides92b]).

```
cellular_object: G11;
data:
        exposed_states : < null, [12-15] >;
        hidden_states : < 7-11>;
        method_sel_states : < null, ALL_TRIED, LOCK>;
       connections : < con1, con2, con3, con4, con5, con6>;
        states_accepted :  < [0..24], rooms, commands, data_types>;
                           / rooms and commands are complex data types
       neighbours : < G7, G8, G9, G10, G12, G17, G11d1, G11d2,
                      G11d3, G11com1, G11com2>;
            / neighbours are  ordered anti-clockwise; by
             /d are identified the associated with the cell data cells and by
             /com  the associated command cells.
        neighbour_vector : <G7(1), G8(2), G9(3), G10(4), G12(1), G17C(2)>

methods:
         check_update_state;
        expose;
        propose;
        apply_filter;                  / for group leader objects
        check_majority_act;            / if voting is used

method_selection_function(exposed_states, method_sel_states, states
                                                        _accepted);
     if method_sel_state = null then check_update_state;
            ..............................
    if exposed_state = null & method_sel_state = null then propose;
            ..............................
    if method_sel_state = ALL_TR then expose;
            ..............................
```

Figure 3: Cellular object for real-time meeting -scheduling application

4. CONCLUSIONS

Cellular objects have been developed in order to execute on simple connected processing units. They have a simple communication mechanism that supports large number of connections for fine- to medium grain parallel architectures. The structure of the cellular object graph can dynamically be self-reconfigured at run-time. In addition to acting as a modelling and development medium (active data structure) for various symbolic large-scale applications in business and industry, cellular object graphs, themselves, can be used as a paradigm for the development of novel massively parallel architectures for AI.

REFERENCES

[Adamides92a] Adamides, E.D. and Bonvin D., "A Cellular Control Architecture", Proc. IFAC Intelligent Manufacturing, Dearborn, MI, 1992.

[Adamides92b] Adamides, E.D., "Modelling Real-time Distributed Meeting Scheduling with Cellular Objects", submitted to Software Engineering Journal (1992).

[Adamides92c] Adamides, E.D., Tsalides, Ph. and Thanailakis, A. "Hierarchical Cellular Automata Structures", Parallel Computing, 18(1992), pp 517-524.

[Agha86] Agha, G.A., "ACTORS: A Model of Concurrent Computation in DIstributed Systems", The MIT Press, (1986).

[Benveniste91] Benveniste, A. and Berry, D., "The Synchronous Approach to Reactive and Real-Time Systems", Special Section on Another Look at Real-Time Programming", Proc. IEEE, Vol. 79, No.9, (1991), pp 1270-1282.

[Chin91] Chin, R.S. and Chanson, S.T., "Distributed Object-based Programming Systems", ACM Comp. Surv., Vol. 23, No. 1, (1991), pp 91-24.

[Hillis85] Hillis, W.D., "The Connection Machine", The MIT Press, 1985.