Time, Communication and Synchronisation in an Agent-Based Programming Language

Rafael Ramirez
National University of Singapore
Information Systems and Computer Science Department
Lower Kent Ridge Road, Singapore 119260
rafael@iscs.nus.sg

Abstract

In this paper, we describe an approach to the representation, specification and implementation of multi-agent real-time systems. The approach is based on the notion of concurrent object-oriented systems where processes are represented as objects. As argued in [3], this is a highly suitable base for extension to distributed AI and multi-agent application platforms. In our approach, the behaviour of an agent (its safety properties and time requirements) is declaratively stated as a set of temporal constraints among events which provides great advantages in writing multi-agent systems and manipulating them while preserving correctness. The temporal constraints have a procedural interpretation that allows them to be executed, also concurrently. In this way, the specification of an agent behaviour is directly executed, thus eliminating the need to verify that the implementation satisfies its specification. The approach also provides a framework in which algorithms for a variety of concurrent programming paradigms may be expressed, compared and manipulated, and can be used as the basis for a development methodology for multi-agent real-time systems.

1 Introduction

Multi-agent systems are being developed and applied in a variety of areas, from traditional AI, through operating systems and data communications, to robotics and concurrent language design. However, although the term 'agent' is widely used among researchers in these areas, it is difficult to produce a single universally accepted definition [17]. Furthermore, in spite of the amount of research carried out on the subject, there seems to be a lack of development methods and verification techniques for multi-agents systems. Also, agent-based programming languages, and in general concurrent programming languages, usually focus on *qualitative* aspects, i.e. defining what constitutes a legal execution of a program. This is unsuitable for real-time programming. The correctness of real-time systems requires that both *qualitative* and *quantitative* temporal requirements to be observed during program execution.

In this paper we describe an approach to the representation, specification, development and implementation of multi-agent real-time systems. The approach is based on the notion of concurrent object-oriented real-time systems. Each agent is explicitly described as a partially ordered set of events and executes its events in the specified order. The partial order is defined by a set of temporal constraints and agent synchronisation and communication are handled by shared events. Both, agent behaviour (its safety properties and time requirements) and synchronisation are declaratively stated which provides great advantages in writing multi-agent systems and manipulating them while preserving correctness. Agent specifications have a procedural interpretation that allows them to be executed, also concurrently. In this way, the specification of an agent behaviour is directly executed, thus eliminating the need to verify that the implementation satisfies its specification. The approach also provides a framework in which algorithms for a variety of concurrent programming paradigms may be expressed, compared and manipulated, and can be used as the basis for a development methodology for multi-agent real-time systems. First, the system can be specified in a simple manner, and then this specification may be incrementally strengthened and divided into agents that communicate using the intended target paradigm.

This work is strongly related to Tempo++ ([12], [11]), a concurrent programming language which combines logic programming and object-oriented programming, and to its real-time extensions [13], [14]. In Tempo++, objects are explicitly described as partially ordered set of events. Events are executed in the specified order, but their execution times are only *implicit*. Actions may be associated with events in order to specify the object computation, and inter-object synchronisation and communication is handled through shared events. Tempo++ has been extended to support real-time by making event execution times *explicit* and by allowing the specification of the system time requirements in terms of relations among these execution times. Here, we will motivate the use of the described computational model by showing it can be applied to the representation, specification, implementation and development of multi-agent real-time systems.

Recently, a number of languages intended for programming multi-agent systems have been proposed. However, most of them are not based on a logical theory. For instance, languages such as April [8], AgentSpeak [16] and, more recently, the work by Nielsen and Agha [9] do not seem to be based in any logical theory. Closer to our work is the work by Shoham [15] and Fisher [2]. Shoham proposed a framework which consists of a logical system for defining the mental state of agents, together with a language for programming those agents. Although the programming language provides support for the concepts in the logical system, the formal link between the language and the logic is not clear. The work reported in [2] is perhaps the closest, but the approach presented there makes a clear distinction between agents, expressed in temporal logic, and the interface between them, which is handled by a specific broadcast communication mechanism. Hence, it is not intended for expressing concurrent algorithms in a general way. Also, although the approach provides a succinct and natural way of expressing the desired qualitative temporal requirements of agents, it cannot refer to metric time and, hence, is insufficient for the specification of the quantitative temporal requirements of agents, i.e. it is unsuitable for real-time programming.

Section 2 introduces the core concepts of our approach to multi-agent real-time programming, namely events, precedence constraints and real-time, and outlines a methodology for the development of multi-agent real-time systems. Section 3 extends these concepts by adding practical programming features such as data structures and operations on them as well as supporting object-oriented programming. Finally, Section 4 summarizes our approach and its contributions as well as some areas of future research.

2 Events, constraints and real-time

2.1 Events and precedence

Many researchers, e.g. [6], [10], have proposed methods for reasoning about temporal phenomena using partially ordered sets of events. Our approach to the specification and implementation of multi-agent systems is based on the same general idea. We propose a language in which agents are explicitly described as partially ordered sets of events. The event ordering relation $X < Y$, read as "X precedes Y", is the main primitive predicate in the language (there are only two primitive predicates), its domain is the set of events, and is defined by the following axioms (the last axiom is actually a corollary of the other three):

$$\begin{array}{l}\forall X \forall Y \forall Z (X < Y \wedge Y < Z \rightarrow X < Z) \\ \forall X \forall Y (time(Y, eternity) \rightarrow X < Y) \\ \forall X (X < X \rightarrow time(X, eternity)) \\ \forall X \forall Y (Y < X \wedge time(Y, eternity) \rightarrow \\ \qquad\qquad\qquad\qquad time(X, eternity))\end{array}$$

The meaning of predicate $time(X, Value)$ is "event X is executed at time $Value$" and $eternity$ is interpreted as a time point that is later than all others. Events are atomically executed in the specified order (as soon as its predecessors have been executed) and no event is executed for a variable that has execution time $eternity$. $Long$-$lived$ agents (agents comprising a large or infinite set of events) are specified allowing an event to be associated with one or more other events: its offsprings. The offsprings of an event E are named $E + 1$, $E + 2$, etc., and are implicitly preceded by E, i.e. $E < E + N$, for all N. The first offspring $E + 1$ of E is referred to by $E+$. Syntactically, offsprings are allowed in queries and bodies of constraint definitions, but not in their heads. A formal definition of for event offsprings can be given by the following axioms:

1. $p(E_1, \ldots, E_n, X + N, F_1, \ldots, F_m) \leftrightarrow$
   $(\exists Y\ offs(X, N, Y) \wedge X < Y \wedge$
   $p(E_1, \ldots, E_n, Y, F_1, \ldots, F_m))$
2. $offs(X, N, Y) \wedge offs(X, N, Z) \rightarrow Y = Z$

where $offs(X, N, Y)$ indicates that event Y is the Nth offspring of event X.

Example 1. Consider a producer and consumer system where the producer component goes through the stages $p, s, p, s, p, \ldots$ where p and s respectively stand for occurrences of $produce$ and $send$, and the consumer component goes through $r, c, r, c, r, \ldots$ where r and c respectively stand for occurrences $receive$ and $consume$. The two components are linked by an infinite buffer. In the language, the producer's behaviour may be specified by the following query ("," denotes conjunction)

$$\leftarrow P < S, S < P+, P+ < S+, S+ < P++, \ldots$$

where P and S represent occurrences of events $produce$ and $send$, respectively ($P+$, $S+$, $P++$, $S++$, $\ldots$ represent later occurrences of these events, e.g. $P+$ represents the second occurrence of event $produce$). Such behaviour can also be recursively specified by the query

$$\leftarrow P <\!\!* S, S <\!\!* P+.$$

where the user-defined constraint $X <\!\!* Y$ is defined as

$$X <\!\!* Y \leftarrow X < Y, X+ <\!\!* Y+.$$

In words, two events X and Y are in the relation $<*$ if X precedes Y and their first offsprings are $<*$-related. Similarly, the consumer's behaviour may be specified by the query

$$\leftarrow R <\!\!* C, C <\!\!* R+,$$

and the buffer's behaviour by

$$\leftarrow S <\!\!* R.$$

The specification of the complete system is thus the conjunction of its components, i.e.

$$\leftarrow P <\!\!* S, S <\!\!* P+, S <\!\!* R, R <\!\!* C, C <\!\!* R+.$$

Long-lived agents may not terminate because body events are repeatedly introduced. Interestingly, an infinitely defined constraint need not cause non-termination: a constraint, whose arguments all have time value $eternity$ will not be expanded. The time value of an event E can be bound to $eternity$ (as a consequence of the axioms defining "$<$") by enforcing the constraint $E < E$. No offsprings of E will be executed. Their time values are known to be bound to $eternity$ since they are implicitly preceded by E and the time value of E is $eternity$. For instance, the behaviour of the producer defined in Example 1 may be modified to stop after producing three items by adding the constraint $P{+}{+}{+} < P{+}{+}{+}$ to its specification.

In the language, disjunction is specified by the disjunction operator ';' (which has lower priority than ',' but higher than '$\leftarrow$'). The clause $H \leftarrow Cs_1; \ldots; Cs_n$ abbreviates the set of clauses $H \leftarrow Cs_1, \ldots, H \leftarrow Cs_n$. In the absence of disjunction, a query determines a unique set of constraints. An interpreter produces any execution sequence satisfying those constraints, it does not matter which one. With disjunction, a single set of constraints must be chosen from among many possible sets, i.e., a single alternative must be selected from each disjunction. Operationally, disjunction is handled as follows: While testing whether an event E is enabled, i.e. it is not preceded by another event, every alternative of each disjunction constraint D is tried. If E is enabled by some but not all alternatives of D (we say E is conditionally enabled), we enable E and reduce D: retain the enabling alternatives and delete the others. If all alternatives of D enable E, E is enabled; if no alternatives enable E, E is disabled. In either case, D is not reduced. For example, given the query

$$\begin{aligned}\leftarrow B < A, C < A, p(A);\\ A < B, C < B, q(B);\\ A < C, B < C, r(C).\end{aligned}$$

event A could be enabled by reducing the disjunction to

$$\leftarrow C < B, q(B); B < C, r(C).$$

and subsequently C could be enabled by reducing the disjunction to a single alternative

$$\leftarrow q(B).$$

A different alternative will be chosen depending on which of the three events is the last to occur.

In the language described above, agents are explicitly described as partially ordered sets of events. Their behavior is specified as logical formulas which define temporal constraints among a set of events. This logical specification of an agent has a well defined and understood semantics and allows for the possibility of employing both specification and verification techniques based on formal logic in the development of multi-agent systems.

2.2 Real-time

Time requirements in the language may be specified by using the primitive predicate $time(X, Value)$. This constrains the execution time of event X by forcing X to be executed at time $Value$. In this way, quantitative temporal requirements (e.g. maximal, minimal and exact distance between events) can be expressed in the language. For instance, maximal distance between two events E and F may be specified by the constraint $max(E, F, N)$, meaning "event E is followed by event F within N time units", and defined by

$$\begin{aligned}max(E, F, N) \leftarrow E < F, time(E, Et),\\ time(F, Ft), Ft \prec Et + N.\end{aligned}$$

where $\prec$ is the usual $less\ than$ arithmetic relationship among real numbers. Thus, maximal distance between families of events, i.e. events and their offsprings, can be specified by the constraint $max*\ (E, F, N)$, meaning "occurrences of events E, $E+$, $E{+}{+}$,... are respectively followed by occurrences of events F, $F+$, $F++$, ... within N time units", and defined by

$$max*(E, F, N) \leftarrow max(E, F, N), max*(E+, F+, N).$$

Minimal and exact distance between two events, as well as other common quantitative temporal requirements, may be similarly specified.

Example 2. Consider a controller designed to handle a gate at a railroad crossing. The complete system consists of three components: the gate controller, the gate itself, and the train. The system's time requirements constrain the train to spend at most 5 units of time to cross the gate area and to approach the gate with a speed which allows 2 units of time to the gate to lower before the arrival of the train. Also, the gate controller must react in less that 1 unit of time to the approach of the train, and the gate should employ at most 1 unit of time to completely lower or raise. The behaviour of the train can be specified as follows:

$$
\begin{array}{l}
train(App, In, Out, Exit) \leftarrow \\
\quad App <\!* In, \\
\quad In <\!* Out, \\
\quad Out <\!* Exit, \\
\quad min\!*\ (App, In, 2), \\
\quad max\!*\ (App, Exit, 5).
\end{array}
$$

$$
\begin{array}{l}
min(E, F, N) \leftarrow \\
\quad E < F, time(E, Et), \\
\quad time(F, Ft), Et + N \prec Ft. \\
min\!*\ (E, F, N) \leftarrow \\
\quad min(E, F, N), \\
\quad min\!*\ (E+, F+, N).
\end{array}
$$

where events *App, In, Out* and *Exit* respectively denote the events of the train approaching the gate, the train entering the gate, the train leaving the gate, and the train leaving the gate area. Constraint $max\!*\ (E, F, N)$ is defined as before. Similarly, the behaviour of the controller can be specified by

$$
\begin{array}{l}
cont(App, Lower, Exit, Raise) \leftarrow \\
\quad App <\!* Lower, \\
\quad Lower <\!* Exit, \\
\quad Exit <\!* Raise, \\
\quad edist\!*\ (App, Lower, 1), \\
\quad max\!*\ (Exit, Raise, 1).
\end{array}
$$

$$
\begin{array}{l}
edist(E, F, N) \leftarrow \\
\quad E < F, time(E, Et), \\
\quad time(F, Ft), Ft = Et + N. \\
edist\!*\ (E, F, N) \leftarrow \\
\quad edist(E, F, N), \\
\quad edist\!*\ (E+, F+, N)
\end{array}
$$

where events *Lower* and *Raise* respectively correspond to the events of the gate starting to lower and the gate starting to raise. Finally, the specification of the gate is given by

$$
\begin{array}{l}
gate(Lower, Down, Raise, Up) \leftarrow \\
\quad Lower <\!* Down, \\
\quad Down <\!* Raise, \\
\quad Raise <\!* Up, \\
\quad max\!*\ (Lower, Down, 1), \\
\quad min\!*\ (Raise, Up, 1), \\
\quad max\!*\ (Raise, Up, 2).
\end{array}
$$

where events it Down and *Up* respectively denote the events of the gate arriving to the down position, and the gate arriving to the up position. The specification of the complete railroad crossing system is the conjunction of its components, i.e.,

$$
\begin{array}{l}
\leftarrow train(App, In, Out, Exit), \\
\quad cont(App, Lower, Exit, Raise), \\
\quad gate(Lower, Down, Raise, Up).
\end{array}
$$

2.3 Development of multi-agent systems

One methodology for program development consists of the refinement of specifications, i.e. adding detail to specifications. Given a problem, a solution strategy is proposed and specified. Usually the strategy is broad and general. Then the strategy specification is proved to actually solve the problem. Sometimes the solution to the problem is required to suit a particular target architecture, in which case the strategy is narrowed by refining the specification. Specification refinements are required to be proved correct, i.e. it is necessary to prove that the specification proposed indeed refines a specification proposed at an earlier step. Once a problem has been specified in extensive detail, the construction of the program begins. Finally, it is verified that the implementation satisfies its specification.

In our approach, the specification of the behaviour of the agents in the system is a program, i.e. it is possible to directly execute the specification. Thus, a program P may be transformed into a program that logically implies P (in [4] some transformations rules that can be applied to programs are presented). The derived program is guaranteed to have the same safety properties as the original one, though its progress properties may differ, e.g. one may terminate and the other not. The program may be incrementally strengthened by introducing timing constraints to specify the system time requirements. Finally, the program can be turned into a concurrent one by grouping constraints into agents. This final step affects neither the safety nor the progress properties of the algorithm, provided that the restriction that each event E can be conditionally enabled in at most one agent is observed.

3 Constraints and agents

Usually, agents are described as encapsulated entities with some artificial intelligence features. We believe that there is practically no distinction between an agent, as defined above, and an object, as considered in concurrent object-oriented systems. Although some researchers (e.g. [7]) distinguish agents from objects arguing that agents have control over their own execution while object do not, here, we will assume that there is no real distinction between an agent and an object and that a multi-agent system is simply a system consisting of concurrently executing objects.

Our approach to the specification and implementation of multi-agent real-time systems is based on an extension to the logic presented in the previous section. The logic is extended by adding data structures and operations on them, by allowing values to be assigned to events for inter-agent communication and by supporting object-oriented programming. In the following, we incrementally present the programming language which results from these extensions. A detailed discussion of the ideas behind the extensions can be found in [12].

3.1 Encapsulated constraints

Object-oriented programming introduces a guiding principle for writing programs which provides a simple but powerful model for representing programs as cooperative collections of computational entities. Each of these entities represents an instance of some *class*, which in turn are organised hierarchically by an *inheritance* relationship.

Our language supports object-oriented programming by allowing a class to encapsulate a set of constraints, specified by a constraint query, together with the related constraint definitions, specified by a set of clauses, in such a way that it describes a set of potential run-time agents. The constraint query defines a partial order among a set of events, and the constraint definitions provide meaning to the user-defined constraints in the query. Both the query and definitions are local to the class. Each of the class run-time agents corresponds to a concurrent object. The name of a class may include variable arguments in order to distinguish different instances of the same class. Events appearing in the constraint query of an agent implicitly belong to that agent. If an event is shared between several agents (it belongs to two or more agents), it cannot be executed until it has been enabled by all agents that share it. For example, consider the producer described earlier in the paper. A class `Producer(I)` specifying the same behavior may be defined by the following structure (constraint $<*$ is defined as before and it is assumed to be globally defined):

```
Producer(I)
constraints
   prod(P(I),S(I)).

   prod(P,S) <- P<*S, S<*P+.
endclass
```

The class `Producer(I)` may be used as a pattern to create producer agents. When created, each agent receives a copy of the constraints of the class of which it is an instance. In the case that a class constraint query contains events of the form $X(I)$, where I is an argument in the class name, the value of I is passed to these events. It is assumed that instances of a same class are created using different argument values (otherwise they would be identical agents), thus they do not share events of the form $E(I)$, and sharing of such events is restricted to instances of different structures.

Synchronization

Inter-agent synchronisation may be handled either by one of the synchronised agents or by an extra agent. For example, if we specify the consumer and buffer components described in the previous section as follows,

```
Consumer(I)
constraints
   cons(R(I),C(I)).

   prod(R,C) <- R<*C, C<*R+.
endclass

Buffer(I)
constraints
   buf(S(I),R(I)).

   buf(S,R) <- S<*R.
endclass
```

the consumer is not allowed to execute event R (receive) until it is enabled by both itself and the buffer, i.e. the two agents sharing R.

3.2 Events and actions

As mentioned before, the agent behaviour is specified as a partially ordered set of events. In order to specify the agent computation, actions may be associated with these events. The representation and manipulation of data is handled by these actions. In principle, these actions may be specified in any language. However, our current implementation of the language assumes an action to be a definite goal. In order to execute an event, the goal associated with it (if any) has to be solved first.

As pointed before, most interesting agents comprise a large or infinite set of events, which it is not feasible to name individually in a specification. Thus, in the context of actions, agents with a large or infinite set of events may need a large or infinite set of event-action mappings. The language provides constructs to allow sets of such mappings to be specified for families of events, i.e. an event and its offsprings, as well as to associate different actions to different offsprings in the same tree.

Agents may also encapsulate private information on how to solve the actions (goals) associated with events. This information consists of predicate definitions which provide meaning to the event actions. The predicate definitions are Horn clauses extended by allowing class names as part of their bodies. This extension to the usual Horn clause syntax permits the dynamic creation of agents. Both, actions and predicate definitions are specified by including special sections in the class structure. Note that both the goals representing event actions and the predicate definitions providing meaning to the actions have no semantic overlap with the event logic, i.e. the logic which specifies the partial order among the events.

3.3 Agent communication

In the language, agents communicate via shared events' values. Shared events represent communication channels and values assigned to them represent messages. An event E may have a data value associated with it, referred as $val(E)$. The term $val(E)$ may be thought of as a logical variable which may be instantiated with the value associated with event E. Value assignments must respect the following restrictions:

1. the action of assigning a value to an event E must be executed before event E is enabled, and its value must be accessed only after E has been executed.

2. the value of an event is allowed to be instantiated by at most one object.

The reason for these restrictions is the need to ensure the right order in assigning and accessing event values, i.e. to avoid accessing an uninstantiated event value, and the attempt to instantiate an already instantiated event value.

For instance, in the case of the producer and consumer system previously presented, the buffer component, in addition to provide synchronisation between producer and consumer, has to transfer information from the former to the latter. This may be done by the introduction of actions in the producer, buffer and consumer. The information is produced (event P in the producer), it is assigned to shared event S (shared by producer and buffer). In fact, producer generates values for events $S, S+, S++$, etc. In this way information is made available to the buffer, which in turn assigns these values to events R, $R+$, $R{+}{+}$, etc., respectively. These events are shared by the buffer and the consumer and thus, their values are accessible to the consumer.

3.4 Inheritance

One of the main advantages of the object-oriented approach is inheritance. Objects are organised in a hierarchy where an object inherits properties from its ancestors. The language described here supports both single and multiple inheritance by using *inherits* declarations. An agent can *partially* inherit another agent, i.e. an agent can inherit either another agent's temporal constraints, actions or actions definitions. Thus, inheritance of concurrency issues and inheritance of code are independently supported. This allows an agent to have its synchronisation scheme inherited from another agent while defining its own code, or vice versa, or even inherit its synchronisation scheme and code from two different agents. A full discussion of inheritance in the language can be found in [12].

3.5 Multi-paradigm approach

Our language appears to add to the proliferation of concurrent programming paradigms: processes (agents) communicate via a new medium, *shared events*. However, our objective is rather to simplify matters by providing a framework in which algorithms for a variety of concurrent programming paradigms can be expressed, derived, and compared. Among the paradigms we have considered are synchronuos message passing, asynchronous message passing and shared mutable variables.

In synchronous message passing the exchange of a message is an *atomic* action requiring the participation of both the sending process and the receiving process. If the sending process S is ready to send a message but the receiving process R is not ready to receive, then S is blocked, and conversely, if R is ready to receive before S is ready to send, then R blocks. The two processes synchronise when they communicate with each other. In our approach, an equivalent algorithm can be derived. A program in which each event is shared by at most two agents simulates a program using point-to-point synchronous message passing. The shared event represents a message exchange between a sender and receiver, which takes place when both are ready (they have both enabled the event). If the event is conditionally enabled in one agent (i.e. the event appears inside a disjunction and is enabled by some disjuncts and disabled in all the others), that agent corresponds to the receiver, otherwise there is no distinction. Local (unshared) events represent the internal actions of an agent.

In asynchronous communication the sending process is allowed to send a message and continue without blocking. Except for the fact that a message has to be sent before it can be received, there is no time connection between the execution of the sending process and the receiving process. Here, asynchronous message passing can be simulated by a program in which each shared event appears on the right of a $<$ constraint in exactly one agent: the sender. Other agents sharing the event are receivers. The shared event then corresponds to the asynchronous sending of a message: it can be delayed only by the single sender agent.

Algorithms in which processes communicate by shared variables are well suited to shared memory parallel architectures. In such algorithms, processes communicate by instantiating and reading shared mutable variables (variables that can be destructively modified by several processes). Our approach can model programs in which agents communicate via shared mutable variables. Each shared variable is represented by a var constraint, contained in its own agent, and events can be shared only between a var agent and another agent. A complete description of how to express, derive, and compare algorithms for a variety of concurrent programming paradigms can be found in [5].

3.6 Agent execution

In the language, agents are specified as a set of temporal constraints among events and a set of logical goals associated to such events. The temporal constraints have a procedural interpretation which allows the events to be executed in the specified order, triggering the execution of the corresponding goals. The key point is that the procedural interpretation actually $executes$ the specified agents, rather than merely returning answer substitutions for the execution times of the events in the query. This is,

1. Instead of computing an execution time t for event E, an interpreter of the language executes event E at time t (its execution may involve executing an action, i.e. solving a logical goal). No event is executed for a variable that has execution time $eternity$. This way, events are executed in the specified order and according to the system timing requirements.

2. The solution is constructed incrementally in ascending time order. That is, an event is executed as soon as its predecessors have been executed and taking in account its timing requirements, without waiting for a complete solution. This yields useful results even if the set of events is large or infinite. The language interpreter uses a constraint set CS containing the constraints still to be satisfied, and a try list TL containing the events that are to be tried but have not yet been executed. The interpreter constantly takes events from TL and checks if they are $enabled$, i.e. if they are not preceded by any other event (according to CS), and the timing constraints on their execution times are satisfiable, in which case they are executed. The order in which the events are tried is determined by the timing constraints in CS.

3. *Fairness* in the language is implicitly guaranteed: every event that becomes enabled will eventually be executed. This is implemented by making the try list TL a FIFO queue.

4 Conclusions

We have described an approach to the representation, specification and implementation of multi-agents real-time systems. The approach is based on the notion of concurrent object-oriented systems where processes are represented as objects. As it has been argued in the past, this is a highly suitable base for extension to distributed AI and multi-agent application platforms. In the approach, each agent is explicitly described as a partially ordered set of events and executes its events in the specified order. The partial order is defined by a set of temporal constraints and agent synchronisation and communication are handled by shared events. Agent behaviour (safety properties and time requirements) are declaratively stated which provides great advantages in writing multi-agent systems and manipulating them while preserving correctness. The specification of the behaviour of an agent has a procedural interpretation that allows it to be executed, also concurrently, thus eliminating the need to verify that the implementation satisfies its specification. Our approach can also be used as the basis for a development methodology for multi-agent systems. First, the system can be specified in a simple manner, and then this specification may be incrementally strengthened and divided into agents that communicate using the intended target paradigm. The language supports both single and multiple inheritance. An agent can *partially* inherit another agent, i.e. an agent can inherit either another agent's temporal constraints, actions or actions definitions. Thus, inheritance of concurrency issues and inheritance of code are independently supported.

Current status. A prototype implementation of the complete language has been written in Prolog, and used to test the code of a number of applications. Among the applications is the implementation of an airline reservation system in which the users of the system are modeled as agents who interact with each other by consulting and updating an airline database. The discussion of the applications is out of the scope of this paper.

Future work. In the language presented, the action associated with an event can, in principle, be specified in any programming language. Thus, different types of languages, such as the imperative languages, should be considered and their interaction with the model investigated.

Events are considered atomic. Instead of being atomic, they could be treated as time intervals during which other events can occur ([1] and [6]). Such events can be further decomposed to provide an arbitrary degree of detail. This could be useful in deriving programs from specifications: if event F is included within E, F will automatically inherit all of E's predecessors and successors. To do this, we could give each event a $beginning$ and end time value, and define four primitive precedence constraints: "E end before F begins" (the existing '$<$'), "E begins before F begins", "E ends before F ends", and "E begins before F ends".

In programs, all safety properties are explicitly stated. However, the progress properties of programs remain implicit. It would be desirable to be able to express these properties explicitly in a program, but so far we have not devised a way to do that.

Agent behaviour is specified as logical formulas which define temporal constraints among a set of events. This logical specification of an agent has a well defined and understood semantics and we are planning to look carefully into the possibility of employing both specification and verification techniques based on formal logic in the development of multi-agent systems.

References

[1] J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 11(26):832–843, 1983.

[2] M. Fisher. Representing and executing agent-based systems. *Intelligent Agents ECAI'94 workshop*. LNCS 890, pp.307-323, Springer-Verlag, 1994.

[3] L. Gasser and J.P. Briot. Object-based concurrent programming and DAI. *Distributed Artificial Intelligence: Theory and Praxis*, Kluwer, 1992.

[4] S. Gregory. Derivation of concurrent algorithms in tempo. *LOPSTR95: Fifth International Workshop on Logic Program Synthesis and Transformation*, 1995.

[5] S. Gregory and R. Ramirez. Tempo: a declarative concurrent programming language. *Proc.of the ICLP (Tokyo, June), MIT Press*, 1995.

[6] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, (4):67–95, 1986.

[7] T. Maruichi, M. Ichikawa and M. Tokoro. Modelling autonomous agents and their groups *Decentralized AI 2 - Proceedings of the 2nd European Workshop on modelling autonomous objects and Multi-Agent Worlds*, Elsevier/North Holland 1991.

[8] F. McCabe and K. Clark. April - agent process interaction language. *Intelligent Agents ECAI'94 workshop*. LNCS 890, pp.324-340, Springer-Verlag, 1994.

[9] B. Nielsen and G. Agha. Semantics for an Actor-Based Real-Time Language. *Proceedings of the 4th International Wrokshop on Parallel and Distributed Real-Time Systems*, 1996.

[10] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 1(15):33–71, 1986.

[11] R. Ramirez. Concurrent object-oriented programming in tempo++. *Proceedings of the Second Asian computing Science Conference (Asian'96), LNCS 1179*, pages 244–253, 1996.

[12] R. Ramirez. A logic-based concurrent object-oriented programming language. *PhD thesis, Bristol University*, 1996.

[13] R. Ramirez. Towards declarative concurrent real-time programming *Proceedings of the Australasian Conference on Parallel and Real-Time Systems (Part'97), Springer-Verlag*, pages 262–273, 1997.

[14] R. Ramirez. A logical approach for specification and execution of concurrent real-time systems *Proceedings of the 1997 International Workshop on Real-Time Computing Systems and Applications (RTCSA'97), IEEE Computer Society Press*, 1997.

[15] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1), pp.51-92, 1993.

[16] D. Weerasooriya, A. Rao and K. Ramamohanarao. Design of a concurrent agent-oriented language. *Intelligent Agents ECAI'94 workshop*, LNCS 890, Springer-Verlag, 1994.

[17] M. Wooldridge and N.R. Jennings. Intelligent Agents. *ECAI'94 workshop*, LNCS 890, pp.1-39, Springer-Verlag, 1994.