Representing and executing real-time systems

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 real-time systems. The approach is based on the notion of concurrent object-oriented systems where processes are represented as objects. In our approach, the behaviour of an object (its safety properties and time requirements) is declaratively stated as a set of temporal constraints among events which provides great advantages in writing concurrent real-time systems and manipulating them while preserving correctness. The temporal constraints have a procedural interpretation that allows them to be executed, also concurrently. Concurrency issues and time requirements are separated from the code, minimizing dependency between application functionality and concurrency/timing control.

1 introduction

The study of concurrent and distributed programming has become dramatically more important in recent years, thanks largely to the rapidly increasing use of computer networks. However, programming distributed systems is an inherently more difficult task than conventional sequential programming, in respect of both correctness (to achieve correct synchronization) and efficiency (to minimize slow interprocess communication). While in traditional sequential programming the problem is reduced to make sure that the program's final result (if any) is correct and that the program terminates, in concurrent programming it is not necessary to obtain a final result but to ensure that several properties hold during program execution. These properties are classified into safety properties, those that must always be true, and progress (or liveness) properties, those that must eventually be true. Partial correctness and termination are special cases of these two properties. To make things even worse, there exists a wide variety of parallel architectures and a corresponding variety of concurrent programming paradigms. For most problems, it is not possible to envisage a general concurrent algorithm which is well suited to all parallel architectures. Real-time systems are inherently concurrent systems which, in addition to usually require synchronisation and communication with both their environment and within their own components, need to execute under timing constraints. We propose an approach which aims to reduce the inherent complexity of writing concurrent real-time systems. Our approach consists of the following:

1. In order to incorporate the benefits of the declarative approach to concurrent real-time programming, it is necessary that a program describe more than its final result. We propose a language based on classical first-order logic in which all safety properties and time requirements of programs are *explicitly* stated.
2. Programs are developed in such a way that they are not specific to any particular concurrent programming paradigm. The proposed language provides a framework in which algorithms for a variety of paradigms can be expressed, derived and compared.
3. Applications can be specified as objects, which provides encapsulation and inheritance. The language object-oriented features produce structured and understandable programs, therefore reducing the number of potential errors.
4. Concurrency issues and timing requirements are separated from the rest of the code, minimizing dependency between application functionality and concurrency control. In addition, concurrency issues and timing requirements can also be independently specified. Thus, it is in general possible to test different synchronisation schemes without modifying the timing requirements and vice versa. Also, a synchronisation/time scheme may be reused by several applications. This provides great advantages in terms of program flexibility, reuse and debugging.

2 Related work

This work is strongly related to Tempo ([8]) and Tempo++ ([17], [16]). Tempo is a declarative concurrent programming language based on first-order logic. It is declarative in the sense that a Tempo program is both an algorithm and a specification of the safety properties of the algorithm. Tempo++ extends Tempo (as presented in [8]) by adding numbers and data structures (and operations and constraints on them), as well as supporting object-oriented programming. Both languages explicitly described processes as partially ordered set of events. Events are executed in the specified order, but their execution times are only *implicit*. Here, we extend Tempo++ to support real-time by making event execution times *explicit* as well as allowing the timing requirements to be specified in terms of relations among these execution times.

Concurrent logic programming is also an important influence for this work. This is because concurrent logic programming languages (e.g. Parlog [6], KL1 [22]) and their object-oriented extensions (e.g. Polka [4], Vulcan [10]) preserve many of the benefits of the abstract logic programming model, such as the logical reading of programs and the use of logical terms to represent data structures. However, although these languages preserve many benefits of the logic programming model, and their programs explicitly specify their final result, important program properties, namely safety and progress properties, remain implicit. These properties have to be preserved by using control features such as modes and sequencing, producing programs with little or no declarative reading [17]. In addition, traditional concurrent logic programming languages do not provide support for real-time programming and thus, are not suitable for this kind of applications.

Concurrent constraint programming languages [19] and their object-oriented and real-time extensions (e.g. [21], [20]) suffer from the same problems as concurrent logic languages. Program safety and progress properties remain implicit. These properties are preserved by checking and imposing value constraints on shared variables in the store. Also, there is no clear separation of programs concurrency control, timing requirements and application functionality. In addition, the concurrent constraint model is most naturally suited for shared memory architectures not being easily adapted to model distributed programming systems.

Our work is also related to specification languages for concurrent real-time systems. It is closer to languages based on temporal logic, e.g. Unity [3] and TLA [13], and real-time extensions of temporal logic, e.g. [12], than process algebras [9], [14] and real-time extensions of state-transition formalisms [2], [5]. Unity and TLA specifications can express the safety and progress properties of concurrent systems but they are not executable. Both formalisms model these systems by sequences of actions that modify a single shared state which might be a bottleneck if the specifications were to be executed.

Section 3 introduces the core concepts of our approach to real-time programming, namely events, precedence constraints and real-time, and outlines a methodology for the development of concurrent real-time systems (to make the paper self-contained there is a small overlap with [8] in Sections 3.1 and 3.3). Section 4 describes how these concepts can be extended by adding practical programming features such as data structures and operations on them as well as supporting object-oriented programming. Finally, Section 5 summarizes our approach and its contributions as well as some areas of future research.

3 Events, constraints and real-time

3.1 Events and precedence

Many researchers, e.g. [11], [15], have proposed methods for reasoning about temporal phenomena using partially ordered sets of events. Our approach to the specification and implementation of real-time systems is based on the same general idea. We propose a language in which processes 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):

$$\forall X \forall Y \forall Z (X < Y \land Y < Z \rightarrow X < Z)$$
$$\forall X \forall Y (time(Y, eternity) \rightarrow X < Y)$$
$$\forall X (X < X \rightarrow time(X, eternity))$$
$$\forall X \forall Y (Y < X \land time(Y, eternity) \rightarrow time(X, eternity))$$

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* processes (processes 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.

Long-lived processes may not terminate because body events are repeatedly introduced. Interestingly, an infinitely-defined constraint need not cause non-termination: a constraint, all whose arguments 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*.

In the language, disjunction is specified by the disjunction operator ';' (which has lower priority than ',' but higher than '$\leftarrow$'). The clause $H \leftarrow Cs1; \ldots; Csn$ abbreviates the set of clauses $H \leftarrow Cs1, \ldots, H \leftarrow Csn$. 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.

In the language described above, processes are explicitly described as partially ordered set of events. Their behavior is specified as logical formulas which define temporal constraints among a set of events. This logical specification of a process 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 concurrent systems.

3.2 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

$$max(E, F, N) \leftarrow E < F, time(E, Et), time(F, Ft), Ft \prec Et + N.$$

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 1. Consider an extension to the traditional producer and consumer system where each product cannot be held inside the buffer longer than time t. The system may be represented by the timed Petri net in Figure 1.

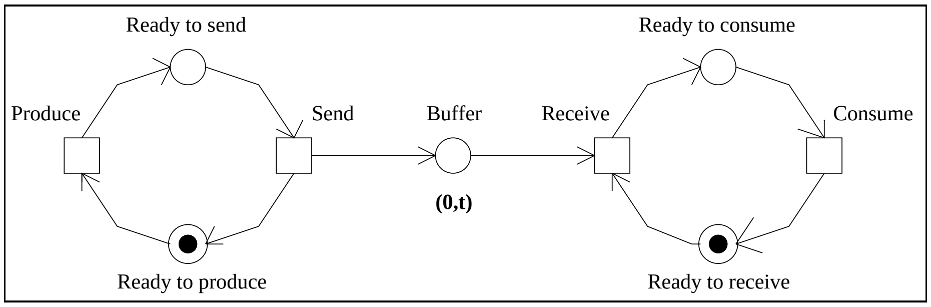

Fig. 1. A producer-consumer system with timed buffer

The producer and consumer components of the system may be respectively specified by $P <\!*S$, $S <\!*P+$ and $R <\!*C$, $C <\!*R+$, where P and S represent occurrences of events *produce* and *send* and R and C represent occurrences of events *receive* and *consume* ($P+$, $S+$, $R+$, $C+$, ... represent later occurrences of these events). The behaviour of the buffer may be specified by

$$tbuf(S, R, T) \leftarrow max(S, R, T), tbuf(S+, R+, T); \; tbuf(S+, R, T).$$

where T is the longest time the buffer can hold a product.

3.3 Development of concurrent real-time systems

In our approach, the specification of the behaviour of the processes 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 [7] 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 processes. This final step affects neither the safety nor the progress properties of the algorithm, provided that some restrictions are observed.

4 Object-oriented programming

Our approach to the specification and implementation of 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 process communication and by supporting object-oriented programming. A detailed discussion of these ideas can be found in [17].

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 objects. The constraint query defines a partial order among a set of events and the timing requirements on their execution times, 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 objects corresponds to a concurrent process. 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 object implicitly belong to that object. If an event is shared between several objects (it belongs to two or more objects), it cannot be executed until it has been enabled by all objects that share it. In order to specify the object computation, actions may be associated with events. The representation and manipulation of data as well as the spawning of new objects is handled by these actions. 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. Objects communicate via shared events' values. Shared events represent communication channels and values assigned to them represent messages.

A novel feature of the approach described here is that an object can *partially* inherit another object, i.e. an object can inherit either another object's temporal constraints or actions. Thus, inheritance of concurrency issues and inheritance of code are independently supported. This allows an object to have its synchronisation scheme inherited from another object while defining its own code, or vice versa, or even inherit its synchronisation scheme and code from two different objects.

Our language appears to add to the proliferation of concurrent programming paradigms: processes (objects) 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.

Execution: Our current implementation 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.

5 Conclusions

We have described an approach to the representation, specification and implementation of concurrent real-time systems. The approach is based on the notion of concurrent object-oriented systems where processes are represented as objects. In the approach, each object 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 object synchronisation and communication are handled by shared events. object behaviour (safety properties and time requirements) are declaratively stated which provides great advantages in writing real-time systems and manipulating them while preserving correctness. The specification of the behaviour of an object has a procedural interpretation that allows it to be executed, also concurrently. Our approach can also be used as the basis for a development methodology for concurrent systems. First, the system can be specified in a perpicuous manner, and then this specification may be incrementally strengthened and divided into objects that communicate using the intended target paradigm. An object can *partially* inherit another object, i.e. an object can inherit either another object'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. The discussion of these 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 [11]). Such events can be further decomposed to provide an arbitrary degree of detail. This could be useful in deriving programs from specifications.

Object behaviour is specified as logical formulas which define temporal constraints among a set of events. This logical specification of an object 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 concurrent real-time systems.

References

1. Allen, J.F. 1983. *Maintaining knowledge about temporal intervals.* Comm. ACM 26, 11, pp.832-843.
2. Alur, R., and Dill, D.L. 1990. *Automata for modeling real-time systems*, in ICALP'90: Automata, Languages and Programming, LNCS 443, pp.322-335. Springer-Verlag.
3. Chandy, K.M. and Misra, J. 1988. *Parallel Program Design.* Addison-Wesley.
4. Davison, A. 1991. *From Parlog to Polka in Two Easy Steps*, in PLILP'91: 3rd Int. Symp. on Programming Language Implementation and LP, Springer LNCS 528, pp.171-182, Passau, Germany, August.
5. Dill, D.L. 1989. *Timing assumptions and verification of finite-state concurrent systems*, in CAV'89: Automatic Verification Methods for Finite-state Systems, LNCS 407, pp. 197-212, Springer-Verlag.
6. Gregory, S. 1987. *Parallel Logic Programming in PARLOG*, Addison-Wesley.
7. Gregory, S. 1995. *Derivation of concurrent algorithms in Tempo.* In LOPSTR95: Fifth International Workshop on Logic Program Synthesis and Transformation.
8. Gregory, S. and Ramirez, R. 1995. *Tempo: a declarative concurrent programming language.* Proc.of the ICLP (Tokyo, June), MIT Press, 1995.
9. Hoare, C.A.R. 1985. *Communicating Sequential Processes*, Prentice Hall.
10. Kahn, K.M., Tribble, D., Miller, M.S., and Bobrow, D.G. 1987. *Vulcan: Logical Concurrent Objects*, In Research Directions in Object-Oriented Programming, B. Shriver, P. Wegner (eds.), MIT Press.
11. Kowalski R., and Sergot, M. 1986. *A Logic-based Calculus of Events*, New Generation Computing, 4, 1, pp.67-95.
12. Koymans, R. 1990. *Specifying real-time properties with metric temporal logic*, Real-time Systems, 2, 4, pp.255-299.
13. Lamport, L. 1994. *The temporal logic of actions.* ACM Trans. on Programming Languages and Systems, 16, 3, pp. 872-923.
14. Milner, R. 1989. *Communication and Concurrency*, Prentice Hall.
15. Pratt, V. 1986. *Modeling concurrency with partial orders*, International Journal of Parallel Programming, 1(15):33-71.
16. Ramirez, R. 1995. *Declarative concurrent object-oriented programming in Tempo++.* In Proceedings of the ICLP'95 Workshop on Parallel Logic Programming Japan, T. Chikayama, H. Nakashima and E. Tick (Ed.).
17. Ramirez, R. 1996. *A logic-based concurrent object-oriented programming language*, PhD thesis, Bristol University.
18. Ramirez, R. 1996. *Concurrent object-oriented programming in Tempo++.* In Proceedings of the Second Asian computing Science Conference (Asian'96), Singapore. LNCS 1179, pp. 244-253. Springer-Verlag
19. Saraswat V. 1993. *Concurrent constraint programming languages*, PhD thesis, Carnegie-Mellon University, 1989. Revised version appears as *Concurrent constraint programming*, MIT Press, 1993.
20. Saraswat V. 1993 et al. *Programming in timed concurrent constraint languages.* In Constraint Programming - Proceedings of the 1993 NATO ACM Symposium, pp. 461-410. Springer-Verlag.
21. Smolka, G. 1995. *The Oz programming model*, Lecture Notes in Computer Science Vol. 1000, Springer-Verlag, pp.324-243.
22. Ueda, K. and Chikayama, T. 1990. *Design of the kernel language for the parallel inference machine.* Computer Journal 33, 6, pp.494-500.