

# Concurrent programming made easy

Rafael Ramirez and Andrew E. Santosa

National University of Singapore  
School of Computing  
S16, 3 Science Drive 2, Singapore 117543  
{rafael, andrews}@comp.nus.edu.sg  
Tel. +65 8742909, Fax +65 7794580

**Abstract.** The task of programming concurrent systems is substantially more difficult than the task of programming sequential systems in respect of both correctness and efficiency. In this paper we describe a constraint-based coordination language for concurrent applications which allows the programmer to *declaratively* state the system safety properties as temporal constraints on specific program points of interest. A system is modeled as a set of processes containing of a sequence of *markers*, denoting the processes points of interest, and a constraint store. Process synchronization is specified incrementally by adding constraints on the markers' visit times into the constraint store. The store, thus, acts as a coordination entity which on the one hand encapsulates the system synchronization requirements, and on the other hand, provides a declarative specification of the system concurrency issues. This provide great advantages in writing concurrent programs and manipulating them while preserving correctness.

**Submission Category:** Technical paper

**Keywords:** Concurrency, constraints, declarative programming.

## 1 Introduction

The task of programming concurrent systems is substantially more difficult than the task of programming sequential systems in respect of both correctness (to achieve correct synchronization) and efficiency. One of the reasons is that it is very difficult to separate the concurrency issues in a program from the rest of the code. Synchronization concerns cannot be neatly encapsulated into a single unit which results in their implementation being scattered throughout the source code. This harms the readability of programs and severely complicates the development and maintenance of concurrent systems. Furthermore, the fact that program synchronization concerns are intertwined with the rest of the code, also complicates the formal treatment of the concurrency issues of the program which directly affects the possibility of formal verification, synthesis and transformation of concurrent programs. We believe that the intrinsic difficulties in writing concurrent systems can be considerably reduced by

- treating the system concurrency issues as orthogonal to the system base functionality, and
- using a declarative formalism to specify the system concurrency requirements.

In this paper we propose a coordination language for writing concurrent applications in which the safety properties of a concurrent program are declaratively stated as temporal constraints. The constraints, on the one hand, encapsulate the system synchronization requirements, and on the other hand, provides a declarative specification of the system concurrency issues. Programs are annotated at points of interest so that the run-time environment monitors and verifies that specific synchronization constraints between the visit times of these points are enforced. The declarative nature of the constraints provides great advantages in reasoning and verifying the correctness of programs [6]. Constraints are *language independent* in that the concurrent application program can be specified in any imperative language such as the C programming language. Furthermore, the language is sufficiently high-level to allow the programmer to glue together separate concurrent threads regardless of their implementation language and application code. The language has a procedural interpretation which is based on the *incremental* and *lazy* generation of constraints, i.e. constraints are considered only when needed to reason about the execution order of current events.

Briefly, constraints can be specified between program points delineated by markers. For a marker  $M$ ,  $time(M)$  (read as “the visit time at  $M$ ”) denotes the time at which the instruction immediately preceding  $M$  has just been completed. In the following, we will refer to  $time(M)$  simply by  $M$  whenever confusion is unlikely. Informally, markers are associated with programs points between instructions, possibly in different threads. Thus, given a pair of markers, constraints can be stated to specify their relative order of execution in all executions of the program.

*Example 1.* Consider a medical application which consists of a program  $P$  that analyzes and diagnoses some data. The program consists of two processes  $P1$  and  $P2$ . In principle,  $P$  can be specified in any imperative programming language and we may represent  $P1$  and  $P2$  as a sequence of functional blocks as in Figure 1.

The points of interest in the program are denoted by markers  $M1, \dots, M5$  between which synchronization constraints may be enforced. A constraint of interest in this system is that whenever the data must be collected before it can be analyzed. This requirement can be captured as a constraint on the times at which markers  $M1$  and  $M5$  are visited. This is, it suffices to specify the constraint

$$M5 < M1$$

in order to satisfy the above requirement.  $X < Y$  can be read as “ $X$  precedes  $Y$ ” or as “the execution time of  $X$  is less than the execution time of  $Y$ ” (we will define  $<$  in more detail in Section 3).

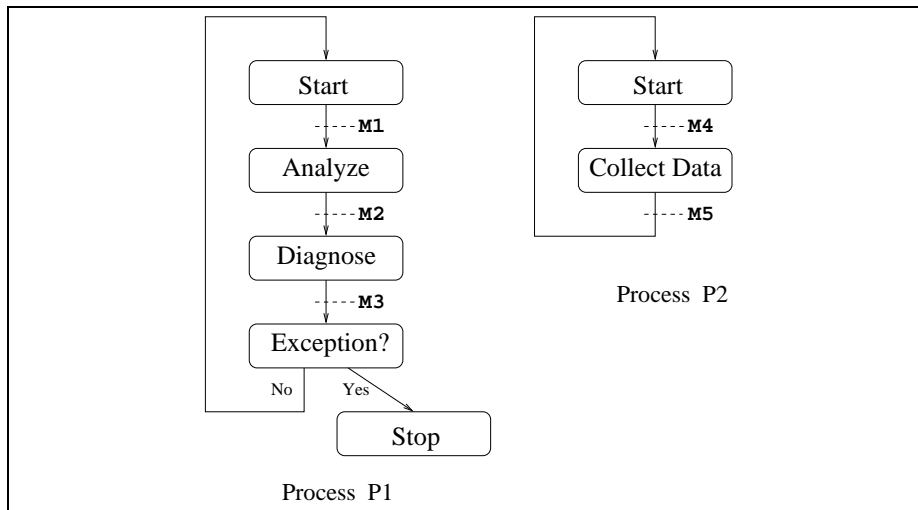


Fig. 1. A medical application

## 2 Related work

### 2.1 Concurrent object-oriented programming

Various attempts have been done by the object-oriented community to separate concurrency issues from functionality. Recently, some researchers have proposed *aspect-oriented programming (AOP)* [9] which encompasses the separation of various program “aspects” (which include synchronization) into codes written in “aspect languages” specific for the aspects. The aspect programs are later combined with the base language program using an *aspect weaver*. In this area, the closest related work is the work by De Volder and D’Hondt [17]. Their proposal utilizes a full-fledged logic programming language as the aspect language. In order to specify concurrency issues in the aspect language, basic synchronization declarations are provided which increase program readability. Unfortunately, the declarations have no formal foundation. This reduces considerably the declarativeness of the approach since correctness of the program concurrency issues directly depend on the implementation of the the declarations.

Closer to our work are *path expressions* (e.g., PROCOL [4]) and constructs similar to *synchronization counters* (e.g., Guide [11] and DRAGOON [2]). These proposals, as ours, differ from the AOP approach in that the specification of the system concurrent issues are part of the final program. Unfortunately, synchronization counters have limited expressiveness since it is not possible to order methods explicitly. Path expressions are more expressive in this respect but it cannot express some important synchronization (e.g., producers-consumers) without embedding guards that increases complexity.

An important issue is that in all of the other proposals mentioned above, method execution is the smallest unit of concurrency. This is impractical in actual concurrent programming where we often need finer-grained concurrency.

## 2.2 Temporal constraints

Most of the previous work on temporal constraints formalisms has concentrated on the specification and verification of real-time systems, e.g. RTL [8] and Hoare logic with time [15]. In addition, there has been research on language notations that consider the synthesis of real-time programs (e.g. TCEL [7], CRL [16], TimeC [12] and Tempo [5]). Tempo is the closest related work. It is a declarative concurrent programming language in which processes are explicitly described as partially ordered set of events. Here, we consider a language in which events are associated with programs points of interest in order to specify the safety properties of a collection of objects. This is done by constraining the execution order of the events by imposing temporal constraints on them.

Section 3 presents the coordination language on which the methodology we propose is based. In Section 4 we give some examples which illustrate how the language may be used to specify the concurrency issues (safety properties) of an application. Section 5 outlines our current implementation, and finally Section 6 summarizes the contributions and indicates some areas of future research.

## 3 Logic programs for concurrent programming

In this section we describe the coordination language which we propose for specifying the time requirements and safety properties of concurrent real-time systems. The main emphasis in the language is for it to be declarative on the one hand, and amenable to execution on the other. Unlike other approaches to concurrent real-time programming, our proposal is concerned only with the specification of the above properties of a system. This is, the application functionality is abstracted away and hence can be written in any conventional imperative programming language.

### 3.1 Events and constraints

Many researchers, e.g. [10, 13], have proposed methods for reasoning about temporal phenomena using partially ordered sets of events. Our approach to concurrent programming is based on the same general idea. The basic idea here is to use a constraint logic program to represent the (usually infinite) set of constraints of interest. The constraints themselves are of the form  $X < Y$ , read as “ $X$  precedes  $Y$ ” or “the execution time of  $X$  is less than the execution time of  $Y$ ”, where  $X$  and  $Y$  are events, and  $<$  is a partial order.

The constraint logic program is defined as follows. Constants range over events classes  $E, F, \dots$  and there is a distinguished (postfixed) functor  $+$ . Thus the terms of interest, apart from variables, are  $e, e+, e++, \dots, f, f+, f++, \dots$

The idea is that  $e$  represents the first event in the class  $E$ ,  $e+$  the next event, etc. Thus, for any event  $X$ ,  $X+$  is implicitly preceded by  $X$ , i.e.  $X < X+$ . We denote by  $e(+N)$  the  $N$ -th event in the class  $E$ . Programs facts or *predicate constraints* are of the form  $p(t_1, \dots, t_n)$  where  $p$  is a user defined predicate and the  $t_i$  are ground terms. Program rules or *predicate definitions* are of the form  $p(X_1, \dots, X_n) \leftarrow B$  where the  $X_i$  are distinct variables and  $B$  a rule body whose variables are in  $\{X_1, \dots, X_n\}$ . A program is a finite collection of rules and is used to define a family of partial orders over events. Intuitively, this family is obtained by unfolding the rules with facts indefinitely, and collecting the (ground) *precedence constraints* of the form  $e < f$ . Multiple rules for a given predicate symbol give rise to different partial orders. For example, since the following program has only one rule for  $p$ :

$$\begin{aligned} & p(e, f). \\ & p(E, F) \leftarrow E < F, p(E+, F+). \end{aligned}$$

it defines just one partial order  $e < f, e+ < f+, e++ < f++, \dots$ . In contrast,

$$\begin{aligned} & p(e, f). \\ & p(E, F) \leftarrow E < F, p(E+, F+). \\ & p(E, F) \leftarrow F < E, p(E+, F+). \end{aligned}$$

defines a family of partial orders over  $\{e, f, e+, f+, e++, f++, e+++ \dots\}$ . We will abbreviate the set of clauses  $H \leftarrow Cs_1, \dots, H \leftarrow Cs_n$  by the *disjunction constraint*  $H \leftarrow Cs_1; \dots; Cs_n$  (disjunction is specified by the disjunction operator ';').

The constraint logic programs have a procedural interpretation that allows a correct specification to be executed in the sense that agents run only as permitted by the constraints represented by the program. This procedural interpretation is based on an incremental execution of the program and a *lazy* generation of the corresponding partial orders. Constraints are generated by the constraint logic program only when needed to reason about the execution times of current events. A description of how this procedural interpretation of constraint logic programs is implemented can be found in [14].

### 3.2 Markers and events

In order to refer to the visit times at points of interest in the program we introduce markers. A marker declaration consists of an event name enclosed by angle brackets, e.g.  $\langle e \rangle$ . Markers annotations can be seen simply as program comments (i.e. they can be ignored) if only the functional semantics of an application is considered. Markers are associated with programs points between instructions, possibly in different threads. Constraints may be specified between program points delineated by these markers. For a marker  $M$ ,  $time(M)$  (read as “the visit time at  $M$ ”) denotes the time at which the instruction immediately preceding  $M$  has just been completed. In the following, we will refer to  $time(M)$  simply by  $M$  whenever confusion is unlikely. Given a pair of markers, constraints

can be stated to specify their relative order of execution in all executions of the program. If the execution of a thread  $T_1$  reaches a program point whose execution time is constrained to be greater than the execution time of a not yet executed program point in a different thread  $T_2$ , thread  $T_1$  is forced to suspend execution. In the presence of loops and procedure calls a marker is typically visited several times during program execution. Thus, in general, a marker  $M$  associated with a program point  $p$  represents an event class  $E$  where each of its instances  $e, e+, e++ \dots$  corresponds to a visit to  $p$  during program execution ( $e$  represents the first visit,  $e+$  the second, etc.).

## 4 Synchronization constraints

In this section we illustrate how the language described is used to specify the concurrency issues (safety properties) of concurrent systems by presenting three examples.

*Example 2.* Consider implementation in Java of a *stack* data type. A basic specification without synchronization code is as follows (ignore for the moment markers `<ai>`, i.e. treat them as comments):

```
class Stack {
    static final int MAX = 10;
    int pos = 0;
    Object[] contents = new Object [ MAX ];

public void print () { <a1>
    System.out.print("[");
    for (int i=0 ; i<pos; i++) {
        System.out.print(contents[i]+" "); }
    System.out.print("]"); <a2> }
public Object peek () { <a3>
    return contents [pos]; <a4> }
public Object pop () { <a5>
    return contents [--pos]; <a6> }
public void push (Object e) { <a7>
    contents [pos++]=e ; <a8> }
}
```

The specification of the class *Stack* with synchronization code added is too complicated to fit comfortably onto a single page. For instance, the declaration of the *peek* method with synchronization is as follows:

```

public Object peek () {
    while(true) {
        synchronized (this) {
            if ((BUSY_pop == 0) && (BUSY_push == 0)) {
                ++BUSY_peek; break; }
            try { wait () ; }
            catch (InterruptedException COOLe ) { } }
        try {
            return contents [pos]; }
        finally {
            synchronized (this) {
                --BUSY_peek;
                notifyAll (); }}}

```

This specifies the safety property that the *peek* method waits until there are no more threads currently executing a *push* or a *pop* method, i.e. mutual exclusion between *peek* and *push* and between *peek* and *pop*. It is clear that the synchronization code completely dominates the source code: almost all of the code for the *peek* method is synchronization code. Furthermore, it is very difficult to formally reason about the correctness of the code.

Similarly, the safety properties that no thread attempts to remove (*pop*) an item from an empty stack and no thread attempts to append (*push*) into a full stack would require coding of additional synchronization code in the *pop* and *push* methods.

These safety property can be elegantly and formally expressed by using temporal constraints as follows. The requirement that the *peek* method waits until there are no more threads currently executing a *push* or a *pop* method may be implemented by

$$\begin{aligned}
 &mutex(a3, a4, a5, a6). \\
 &mutex(a3, a4, a7, a8). \\
 &mutex(X1, X2, Y1, Y2) \leftarrow \\
 &\quad X2 < Y1, mutex(X1+, X2+, Y1, Y2); \\
 &\quad Y2 < X1, mutex(X1, X2, Y1+, Y2+).
 \end{aligned}$$

where  $a3, a4 \dots a8$  are the markers on our initial Java program. The requirement that no thread attempts to remove an item from an empty stack and no thread attempts to append into a full stack may be respectively implemented by

$$\begin{aligned}
 &p(a8, a5). \\
 &p(A, B) \leftarrow A < B, p(A+, B+).
 \end{aligned}$$

and

$$\begin{aligned}
 &p(a6, a7(+MAX)). \\
 &p(A, B) \leftarrow A < B, p(A+, B+).
 \end{aligned}$$

*Example 3.* An example discussed in almost every textbook on concurrent programming (e.g. [1], [3]) is the producer and consumer problem. The problem

considers two types of processes: producers and consumers. Producers create data items (one at a time) which then must be appended to a buffer. Consumers remove items from the buffer (if it is not empty) and consume them, i.e. perform some computation which uses the data item. Here, we consider a system with one producer and one consumer (generalization to the general case is straightforward). Thus, the producer process can be defined by an infinite cycle containing *produce* (producing an item) and *append* (appending the item to the buffer). Similarly, the consumer process can be defined by an infinite cycle containing *remove* (removing an item from the buffer) and *consume* (consuming the item). The producer and consumer may be defined as follows (they have been annotated with markers *p1*, *p2*, *c1* and *c2*).

<pre> <b>Producer</b> <math>\equiv</math> <b>repeat</b>     produce(<i>X</i>);     &lt; <i>p1</i> &gt;     append_item(<i>X</i>);     &lt; <i>p2</i> &gt; <b>forever</b> </pre>	<pre> <b>Consumer</b> <math>\equiv</math> <b>repeat</b>     &lt; <i>c1</i> &gt;     remove_item(<i>X</i>);     &lt; <i>c2</i> &gt;     consume(<i>X</i>) <b>forever</b> </pre>
---	--

If we assume an infinite buffer, the only safety property needed is that the consumer never attempts to remove an item from an empty buffer. This property can be expressed by

```

p(p2, c1).
p(P, C)  $\leftarrow$  P < C, p(P+, C+)

```

In practice however, buffers are finite. A *bounded buffer* can store only a finite number of data elements. Thus, in practice, an extra safety property is that the producer attempts to append items to the buffer only when the buffer is not full. For instance, this safety property for a system with a buffer of size 3 can be expressed by

```

p(c2, p1 + ++).
p(C, P)  $\leftarrow$  C < P, p(C+, P+)

```

*Example 4.* Consider a function *f* from integers to integers with a zero, and suppose we are interested in writing a concurrent program that finds such a zero. The idea is to solve the problem by splitting it into two subproblems that can be solved independently, namely finding a positive and a nonpositive zero. Thus, we can restate the problem as two subproblems: finding *z*<sub>1</sub> such that *z*<sub>1</sub> > 0 and *f*(*z*<sub>1</sub>) = 0 and finding *z*<sub>2</sub> such that *z*<sub>2</sub> ≤ 0 and *f*(*z*<sub>2</sub>) = 0. If *S*<sub>1</sub> and *S*<sub>2</sub> are sequential programs solving these problems, then the parallel execution [*S*<sub>1</sub>||*S*<sub>2</sub>] of *S*<sub>1</sub> and *S*<sub>2</sub> solves the overall problem.

```

S1  $\equiv$  x := 0;
    loop
    x := x + 1;
    if found = false then

```

```

    found := (f(x) = 0)
  else exit
endloop

```

```

S2 ≡ y := 1;
  loop
    y := y - 1;
    if found = false then
      found := (f(y) = 0)
    else exit
  endloop

```

Thus, a solution to the overall problem should be:

$$found := \mathbf{false}; [S_1 || S_2]$$

Unfortunately, although this solution seems intuitively correct it is not. Suppose that  $f$  has exactly one zero, a positive one, and consider an execution of the program where initially, its second component  $S_2$  is activated until it finishes testing the value of  $found$  in the **if** statement. At this moment, only the first component  $S_1$  is executed until it terminates upon finding a zero. The  $S_2$  is activated again and so  $found$  is reset to **false**. Since no other zeroes of  $f$  exist,  $found$  is never reset to **true** and the execution of the program will never terminate. Hence the solution is incorrect. The problem arose because  $found$  could be reset to **false** once it was already **true**, which produced the information that a zero of  $f$  was found to be lost.

One way of correcting this situation is by ensuring that the testing and assignment to  $found$  is done atomically. With this purpose we may annotate  $S_1$  and  $S_2$ ,

<pre> S<sub>1</sub> ≡ x := 0;   loop     x := x + 1;     &lt; e1 &gt;     if found = false then       found := (f(x) = 0)     else exit     &lt; t1 &gt;   endloop </pre>	<pre> S<sub>2</sub> ≡ y := 1   loop     y := y - 1;     &lt; e2 &gt;     if found = false then       found := (f(y) = 0)     else exit     &lt; t2 &gt;   endloop </pre>
---	--

and constraint their execution by

```

mutex(e1, e2, t1, t2).
mutex(E1, E2, T1, T2) ←
  T1 < E2, mutex(E1+, E2, T1+, T2);

```

$$T2 < E1, \text{mutex}(E1, E2+, T1, T2+).$$

## 5 Implementation

A prototype implementation of the ideas presented here has been written using the language Java. Java was used both to implement the constraint language and to write the code of a number of applications.

### 5.1 Architecture

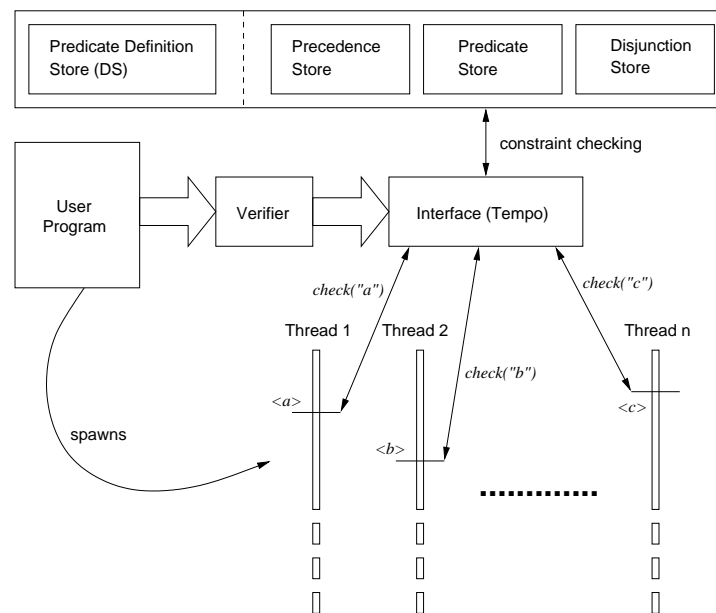


Fig. 2. Implementation architecture

The architecture of our implementation is shown in Figure 2. It consists mainly of three parts:

- The *interface* is an object of class *Tempo* which decides whether or not threads suspend upon reaching a marker during execution. When a thread reaches a marker  $m$ , a request is sent to the interface to determine whether the current event  $e$  associated with  $m$  is *disabled*, i.e. it appears on the right of a precedence constraint  $X < e$ , or *enabled*, i.e. otherwise, wrt the *constraint store*. If  $e$  is found to be disabled, the thread is blocked until  $e$  becomes enabled, otherwise the thread proceeds execution at the instruction immediately after the  $m$ .

- The *constraint store* contains the system synchronization constraints.
- The *user program* is the main program and typically specifies the system synchronization constraints, creates the Tempo object and spawn a number of threads which may contain markers.
- The *verifier* examines the specification of the system synchronization constraints to detect any errors such as infinite loops in predicate definitions, e.g.  $p(X) \leftarrow p(X)$ , before the Tempo object is created.

The overall mechanism is as follows: Once the Tempo object has been created and a set of threads have been spawned, whenever one of the threads reaches a marker in their code, a communication with the constraint store is triggered. Currently the communication is implemented as a request to the Tempo object. The request is of the form  $check(Str)$ , in which  $Str$  is a string denoting marker's identifier (e.g., “p1,” “p2,” “c1,” “c2” in the producers-consumers example above). This request have two differing effects:

1. The request fails causing the invoking thread to suspend if the current event associated with  $Str$  is disabled according to the constraint store.
2. The request succeeds if the current event associated with  $Str$  is enabled according to the constraint store. In this case, the threads proceeds execution at the instruction immediately after the marker and relevant suspended threads are awoken allowing them to re-check the constraint store.

## 5.2 Fairness

*Fairness* is implicitly guaranteed by our implementation. Every event that becomes enabled will eventually be executed (provided that the program point associated with it is reached). This is implemented by dealing with event execution requests in a first-in-first-out basis. Although fairness is provided as the default, users, however, may intervene by specifying priority events using temporal constraints (on how to do this, see [14]). It is therefore possible to specify unfair scheduling.

## 6 Conclusion

We have described a high-level coordination language based on first order logic for expressing synchronization constraints in concurrent programs. In the language, the safety properties of the system are *declaratively* stated as temporal constraints. Programs are annotated at points of interest so that the run-time environment enforces specific temporal relationships between the visit times of these points. Constraints are *language independent* in that the application program can be specified in any conventional concurrent language. The constraints have a procedural interpretation that allows the specification to be executed. The procedural interpretation is based on the incremental and *lazy* generation of constraints, i.e. constraints are considered only when needed to reason about the execution time of current events.

This paper presents work in progress so several important issues are still to be considered. Our implementation is still in a prototype stage, thus several efficiency issues have still to be addressed. In particular, we will focus on how the two key features of incrementality and laziness may be most efficiently achieved. Another important issue is how to deal with progress properties. Currently, constraints explicitly state all safety and timing properties of programs. However, the progress (liveness) properties of programs remain implicit. It would be desirable to be able to express these properties explicitly as additional constraints, but so far we have not devised a way to do that. Future versions may also include deadlock detection feature. We are considering a mechanism that checks user constraints for cycles (e.g.,  $A < B, B < A$ ) whenever a timeout occurred.

## References

1. Andrews, G. R. 1991. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings.
2. Atkinson, C. 1991. *Object-Oriented Reuse, Concurrency and Distribution: An Ada-Based Approach*. Addison-Wesley.
3. Ben-Ari, M. 1990. *Principles of Concurrent and Distributed Programming*. Prentice Hall.
4. Van den Bos, J. and Laffra, C. 1989. *PROCOL: A parallel object language with protocols*. ACM SIGPLAN Notices 24(10):95–112, October 1989. Proc. of OOP-SLA '89.
5. Gregory, S. and Ramirez, R. 1995. *Tempo: a declarative concurrent programming language*. Proc. of the ICLP (Tokyo, June), MIT Press, 1995.
6. Gregory, S. 1995. *Derivation of concurrent algorithms in Tempo*. In LOPSTR95: Fifth International Workshop on Logic Program Synthesis and Transformation.
7. Hong, S. and Gerber, R. 1995. *Compiling real-time programs with timing constraint refinement and structural code motion*, IEEE Transactions on Software Engineering, 21.
8. Jahnaian F. and Mok A. K. 1987. *A graph theoretic approach for timing analysis and its implementation*, IEEE Transactions on Computers, C36(8).
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. 1997. Aspect-oriented programming. In *ECOOP '97—Object-Oriented Programming*, Lecture Notes in Computer Science, number 1241, pp. 220–242, Springer-Verlag.
10. Kowalski, R.A. and Sergot, M.J. 1986. A logic-based calculus of events. *New Generation Computing* 4, pp. 67–95.
11. Krakowiak, S., Meysembourg, M., Nguyen Van, H., Riveill, M., Roisin, C. and Rousset de Pina, X. 1990. *Design and implementation of an object-oriented strongly typed language for distributed applications*. *Journal of Object-Oriented Programming* 3(3):11–22.
12. Leung, A., Palem, K. and Pnueli, A. 1998. *Time C: A Time Constraint Language for ILP Processor Compilation*, Technical Report TR1998-764, New York University.
13. Pratt, V. 1986. Modeling concurrency with partial orders. *International Journal of Parallel Programming* 15, 1, pp. 33–71.
14. Ramirez, R. 1996. *A logic-based concurrent object-oriented programming language*, PhD thesis, Bristol University.
15. Shaw, A. 1989. *Reasoning about time in higher-level language software*, IEEE Transactions on Software Engineering, 15(7).

16. Stoyenko, A. D., Marlowe, T. J. and Younis, M. F. 1996. *A language for complex real-time systems*, Technical Report cis9521, New Jersey Institute of Technology.
17. De Volder, K. and D'Hondt, T. 1999. Aspect-oriented logic meta programming. In *Meta-Level Architectures and Reflection*, Lecture Notes in Computer Science number 1616, pp. 250–272. Springer-Verlag.