

A Methodology for Reliable Concurrent Programming

Rafael Ramirez

Andrew E. Santosa

School of Computing

National University of Singapore

{rafael, andrews}@comp.nus.edu.sg

1. Introduction

Parallel computers and distributed systems are becoming increasingly important. Their impressive computation to cost ratios offer a considerable higher performance than that possible with sequential machines. Yet there are few commercial applications written for them. This is due to two main reasons: (a) concurrency constructs, for expressing synchronization and coordination among processes, is done at a too low level, e.g. message passing or semaphores, and this does not facilitate reasoning about their interaction; (b) these constructs are usually intertwined with the rest of the program, and this means that correctness of concurrent behavior depends on correctness of the entire program. In this paper we propose a methodology for writing reliable concurrent and distributed applications in which

- the system concurrency issues are treated as orthogonal to the system base functionality, and
- a high-level declarative formalism is used to specify the system concurrency requirements.

2. Logic programs for concurrent programming

Many researchers, e.g. [2, 3], 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 (CLP)* [1] in reasoning of event order constraints. The constraints themselves (called *precedence constraints*) are of the form $X < Y$, read as “ X precedes Y ” or “the execution of X precedes the execution of Y ”, where X and Y are events, and $<$ is a partial order.

The CLP is defined as follows. Constants range over events classes E, F, \dots and there is a distinguished (post-fixed) 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 symbol and each t_i is an event, i.e. a ground term. 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 is a rule body restricted to contain variables 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 indefinitely¹, 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 :

$$p(e, f). \\ p(E, F) \leftarrow E < F, p(E+, F+).$$

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

$$p(e, f). \\ p(E, F) \leftarrow E < F, p(E+, F+). \\ p(E, F) \leftarrow F < E, p(E+, F+).$$

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

The CLP has 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

¹In general, *reactive* concurrent programs on which we are focusing, do not terminate.

by the CLP only when needed to reason about the execution times of current events. The full description of the procedural interpretation of CLP can be found in [4]).

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 processes. 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 current time at which the marker is visited. 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 process P_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 process P_2 , process P_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.).

3. The methodology

The treatment of concurrency issues as orthogonal to the rest of the code allows programmers to independently develop one component from the another. A programmer may start implementing a system either by defining a set of abstract processes and their synchronization constraints, or by writing the code for a set of processes, and later identifying the points of interest in their code and introducing synchronization constraint constraints among these points. This is

- **The concurrency-first approach:**
 1. define a skeleton of the system processes.
 2. based on the skeleton, specify the required interprocess synchronization using temporal constraints.
 3. implement the processes functionality.
- **The process-first approach:**
 1. implement a set of sequential processes.
 2. based on this implementation, identify the program points of interest and introduce constraints on the execution order of these points.

An important issue in modern software engineering is reusability. The fact that concurrency issues are separated from the code minimizes dependency between application functionality and concurrency control and introduces the possibility of reusing both software components. It is, in general, possible to test different synchronisation schemes without modifying the code of the operations, and conversely, a synchronisation scheme may be reused by several applications.

We have implemented a web-based tool (<http://www.comp.nus.edu.sg/~rafael/tempo/main.html>) in which users may input the processes skeletons, constraints and predicate definitions using constraints and can visualize the execution of the processes in a simulator panel. Once the user is happy with the specification, the constraints can be saved into the file to be used with the final program.

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

4. Conclusion

We have described a constraint-based methodology for writing concurrent applications. The safety properties of the system are *declaratively* stated as precedence constraints. In this framework, concurrent applications can be developed by firstly specifying a set of sequential processes and later identifying the program points of interest and introducing constraints on the execution order of these points. Alternatively, firstly a skeleton of the system processes can be defined and the required interprocess synchronization specified by temporal constraints and once this is finalized, the processes functionality can be implemented. All this makes programs easier to write, read and verify, and thus greatly improves program reliability.

References

- [1] Jaffar, J., and Lassez, J.-L. 1987. *Constraint Logic Programming*. Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, pp.111-119.
- [2] Kowalski, R.A. and Sergot, M.J. 1986. A logic-based calculus of events. *New Generation Computing* 4, pp. 67–95.
- [3] Pratt, V. 1986. Modeling concurrency with partial orders. *International Journal of Parallel Programming* 15, 1, pp. 33–71.
- [4] Ramirez, R. 1996. *A logic-based concurrent object-oriented programming language*, PhD thesis, Bristol University.