Declarative concurrency in Java

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. We propose a high-level language based on first order logic for expressing synchronization in concurrent object-oriented programs. The language allows the programmer to *declaratively* state the system safety properties as temporal constraints on specific program points of interest. Higher-level synchronization constraints on methods in a class may be defined using these temporal constraints. The constraints are enforced by the run-time environment. We illustrate our language by expressing synchronization of Java programs. However, the general underlying synchronization model we present is *language independent* in that it allows the programmer to glue together separate concurrent threads regardless of their implementation language and application code.

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. In Java, for instance, the problem of writing multi-threaded applications is that synchronization code ensuring data integrity tends to dominate the source code completely. This produces code difficult to understand and modify. To illustrate the problem we will consider in Section 4 the implementation of a *bounded stack* data type. A basic specification without synchronization requires only a few lines of code whereas the specification with synchronization code added is even too complicated to fit onto a single page, its readability is very poor and it is extremely difficult to reason formally about its correctness.

We believe that the system concurrency issues are best treated as orthogonal to the system base functionality. In this paper we propose a first order language in which the safety properties of concurrent programs are declaratively stated as constraints. Programs are annotated at points of interest so that the run-time environment enforces specific temporal constraints between the visit times of these points. The declarative nature of the constraints provides great advantages in reasoning, deriving and verifying concurrent programs [4]. Higher-level constraints on class methods may be defined using the temporal constraints on program points. The constraints are *language independent* in that the application program can be specified in any object-oriented language. The model 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.

Section 2 describes some related work. Section 3 presents the language that we propose for specifying the synchronization constraints of concurrent object-oriented programs. In Section 4 we illustrate the use of the language by implementing a bounded stack data type. Section 5 mentions some implementation issues and finally Section 6 summarizes the contributions and indicates some areas of future research.

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*) [7] 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 [15]. 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 [2]) and constructs similar to *synchronization counters* (e.g., Guide [9] and DRAGOON [1]). 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 [6] and Hoare logic with time [13]. In addition, there has been research on language notations that consider the synthesis of real-time programs (e.g. TCEL [5], CRL [14], TimeC [10] and Tempo [3]). 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.

3 Logic programs for concurrent programming

In this section we describe the logic language which we propose for stating the safety properties of concurrent object-oriented 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 programming, our proposal is concerned only with the specification of the synchronization issues of a system. This is, the application functionality is abstracted away and hence can be written in any conventional object-oriented programming language.

3.1 Events and constraints

Many researchers, e.g. [8, 11], 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[1]. Constants range over events classes $E, F, \ldots$ and there is a distinguished (postfixed) functor $+$. Thus the terms of interest, apart from variables, are $e, e+, e++, \ldots, f, f+, f++, \ldots$. 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 are of the form $p(t_1, \ldots, t_n)$ where p is a user defined predicate and the t_i are ground terms. Program rules are of the form $p(X_1, \ldots, X_n) \leftarrow B$ where the X_i are distinct variables and B a rule body whose variables are in $\{X_1, \ldots, X_n\}$. A program is a finite collection of rules and is used to define a family of partial

[1] For a complete description, see [12]

orders over events. Intuitively, this family is obtained by unfolding the rules with facts indefinitely, and collecting the (ground) 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 $e < f,\ e+ < f+,\ e++ < f++, \ldots$. 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 $\{e, f, e+, f+, e++, f++, e+++\ldots\}$. We will abbreviate the set of clauses $H \leftarrow Cs_1, \ldots, H \leftarrow Cs_n$ by the clause $H \leftarrow Cs_1; \ldots; Cs_n$ (disjunction is specified by the disjunction operator ';').

Example 1. An example discussed in almost every textbook on concurrent programming 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 (represented by event *append*) to a buffer. Consumers remove (represented by event *remove*) items from the buffer and consume them. 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

$p(append, remove)$.
$p(X, Y) \leftarrow X < Y,\ p(X+, Y+)$.

A more practical *bounded buffer* can store only a finite number of data elements. Thus, 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 5 can be expressed by

$p(remove, append(+5))$.
$p(X, Y) \leftarrow X < Y,\ p(X+, Y+)$.

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. `<e>`. 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 $e, e{+}, e{+}{+}\ldots$ corresponds to a visit to p during program execution (e represents the first visit, $e{+}$ the second, etc.).

3.3 Constraints and methods

Sometimes it is more convenient to express the synchronization aspects of an object-oriented program in terms of its methods rather that in terms of specific program points in its code. Thus, we allow in the language higher-level constraints of the form $p(c, m_1, m_2) \Leftarrow q(e_1, e_2, \ldots e_k)$, where m_1 and m_2 are methods in class c and $e_1, e_2 \ldots e_k$ program points (i.e. markers) in the code of either m_1 or m_2. This may be seen as adding some syntactic sugar on top of the base language previously defined.

4 Synchronization constraints

In this section we illustrate how the proposed language is used to specify the concurrency issues (safety properties) of concurrent object-oriented systems by presenting an example: a bounded stack data type.

Consider implementation in Java of a *bounded 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 BoundedStack  {
  static final int MAX = 10;
  int pos = 0;
  Object[] contents = new Object [MAX];

  public Object peek () { <a1>
    return contents [pos]; <a2> }
  public Object pop () { <a3>
    return contents [--pos]; <a4> }
  public void push (Object e) { <a5>
    contents [pos++]=e ; <a6> }
}
```

The specification of the class *BoundedStack* with synchronization code added is shown in the program listing of the next page.

Let us consider, for instance, the declaration of the *peek* method with synchronization as shown in the listing. It 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.

```
public class BoundedStack {
  private static final int MAX = 10;
  private int pos = 0;
  private Object[] contents = new Object[MAX];
  private int activeReaders_ = 0;
  private int activeWriters_ = 0;
  private int waitingReaders_ = 0;
  private int waitingWriters_ = 0;

  private boolean empty() {
    return pos == 0; }
  private boolean full() {
    return pos == MAX; }

  public Object peek() {
    synchronized(this) {
      ++waitingReaders_;
      while (waitingWriters_ == 0 &&
             activeWriters_ == 0) {
        try { wait(); }
        catch (InterruptedException ex) {}
      }
      --waitingReaders_;
      ++activeReaders_;
    }
    try {
      return contents[pos];
    } finally {
      synchronized(this) {
        --activeReaders_;
        notifyAll();
      }
    }
  }
  public Object pop() {
    synchronized(this) {
      ++waitingWriters_;
      while (activeReaders_ == 0 &&
             activeWriters_ == 0 &&
             !empty()) {
        try { wait(); }
        catch (InterruptedException ex) {}
      }
      --waitingWriters_;
      ++activeWriters_;
    }
    try {
      return contents[--pos];
    } finally {
      synchronized(this) {
        --activeWriters_;
        notifyAll();
      }
    }
  }
  public void push(Object e) {
    synchronized(this) {
      ++waitingWriters_;
      while (activeReaders_ == 0 &&
             activeWriters_ == 0 &&
             !full()) {
        try { wait(); }
        catch (InterruptedException ex) {}
      }
      --waitingWriters_;
      ++activeWriters_;
    }
    contents[pos++] = e;
    synchronized(this) {
      --activeWriters_;
      notifyAll();
    }
  }
}
```

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 require coding of additional synchronization code in the *pop* and *push* methods.

These safety properties 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

$$\begin{aligned}
&mutex(a1, a2, a3, a4).\\
&mutex(a1, a2, a5, a6).\\
&mutex(X1, X2, Y1, Y2) \leftarrow X2 < Y1, mutex(X1+, X2+, Y1, Y2);\\
&\qquad\qquad\qquad\qquad\qquad\quad Y2 < X1, mutex(X1, X2, Y1+, Y2+).
\end{aligned}$$

where $a1, a2 \ldots a6$ are the markers on our initial Java program. We may define equivalent higher-level constraints restricting the execution of the *peek*, *pop* and *push* methods by defining:

$$mutex(Stack, peek, push) \Leftarrow mutex(a1, a2, a3, a4)$$
$$mutex(Stack, peek, pop) \Leftarrow mutex(a1, a2, a5, a6)$$

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

$p(a8, a5).$ and $p(a6, a7(+MAX)).$
$p(A, B) \leftarrow A < B,\ p(A+, B+).$ $p(A, B) \leftarrow A < B,\ p(A+, B+).$

5 Implementation

The constraint logic programs have a procedural interpretation that allows a correct specification to be executed in the sense that events are only executed 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 [12].

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 [12]). It is therefore possible to specify unfair scheduling.

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. The discussion of these applications and further details of implementation can be found in a companion paper.

6 Conclusion

We have presented a high-level language for expressing synchronization constraints in concurrent object-oriented applications. In the language, the safety properties of the system are *explicitly* 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. Higher-lever constraints on class methods may also be defined. Constraints are *language independent* in that the application program can be specified in any conventional concurrent object-oriented 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. Atkinson, C. 1991. *Object-Oriented Reuse, Concurrency and Distribution: An Ada-Based Approach.* Addison-Wesley.
2. 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 OOPSLA '89.
3. Gregory, S. and Ramirez, R. 1995. *Tempo: a declarative concurrent programming language.* Proc. of the ICLP (Tokyo, June), MIT Press, 1995.
4. Gregory, S. 1995. *Derivation of concurrent algorithms in Tempo.* In LOPSTR95: Fifth International Workshop on Logic Program Synthesis and Transformation.
5. Hong, S. and Gerber, R. 1995. *Compiling real-time programs with timing constraint refinement and structural code motion*, IEEE Transactions on Software Engineering, 21.
6. Jahnaian F. and Mok A. K. 1987. *A graph theoretic approach for timing analysis and its implementation*, IEEE Transactions on Computers, C36(8).
7. 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.
8. Kowalski, R.A. and Sergot, M.J. 1986. A logic-based calculus of events. New Generation Computing 4, pp. 67–95.
9. 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.
10. 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.
11. Pratt, V. 1986. Modeling concurrency with partial orders. International Journal of Parallel Programming 15, 1, pp. 33–71.
12. Ramirez, R. 1996. *A logic-based concurrent object-oriented programming language*, PhD thesis, Bristol University.
13. Shaw, A. 1989. *Reasoning about time in higher-level language software*, IEEE Transactions on Software Engineering, 15(7).
14. 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.
15. 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.