An Object Model for Multiprogramming

Jayadev Misra

The University of Texas at Austin Austin, Texas 78712, USA

Abstract. We have developed a programming model that integrates concurrency with object-based programming. The model includes features for object definition and instantiation, and it supports concurrent executions of designated methods of the object instances. Yet, the model includes no specific communication or synchronization mechanism, except procedure call. The traditional schemes for communication, synchronization, interfaces among processes and accesses to shared memory can be encoded by objects in our model. Concurrency in the model is transparent to the programmer; the programmer believes that the program executes in a sequential manner whereas the implementation employs concurrent threads to gain efficiency.

1 Introduction

The research described in this document is based on two observations: (1) the applications that will be implemented on networks of processors in the future will be significantly more ambitious than the current applications (which are mostly involved with transmissions of digital data and images), and (2) many of the programming concepts developed for databases, object-oriented programming and designs of reactive systems can be unified into a concise model of distributed programs that can serve as the foundation for designing these future applications.

We have developed a model of multiprogramming, called **Seuss**. Seuss fosters a discipline of programming that makes it possible to understand a program execution as a single thread of control, yet it permits program implementation through multiple threads. As a consequence, it is possible to reason about the properties of a program from its single execution thread, whereas an implementation on a specific platform (e.g., shared memory or message communicating system) may exploit the inherent concurrency appropriately. A central theorem establishes that multiple execution threads implement single execution threads, i.e., any property proven for the latter is a property of the former as well.

A major point of departure in Seuss is that there is no built-in concurrency and no commitment to either shared memory or message-passing style of implementation. No specific communication or synchronization mechanism, except procedure call, is built into the model. In particular, the notions of input/output and their complementary nature in rendezvous-based communication [7, 12] is outside this model. There is no distinction between computation and communication; process specifications and interface specifications are not distinguished. Consequently, we do not have many of the traditional multiprogramming concepts such as, processes, locking, rendezvous, waiting, interference and deadlock, as basic concepts in our model. Yet, typical multiprograms

employing message passing over bounded or unbounded channels can be encoded in Seuss by declaring the processes and channels as the components of a program; similarly, shared memory multiprograms can be encoded by having processes and memories as components. Seuss permits a mixture of either style of programming, and a variety of different interaction mechanisms – semaphore, critical region, 4-phase handshake, etc. – can be encoded as components.

Seuss proposes a complete disentanglement of the sequential and concurrent aspects of programming. We expect large sections of code to be written, understood and reasoned-about as sequential programs. *We view multiprogramming as a way to orchestrate the executions of these sequential programs, by specifying the conditions under which each program is to be executed.* Typically, several sequential programs will execute simultaneously; yet, we can guarantee that their executions would be non-interfering, and hence, each program may be regarded as atomic. We propose an efficient implementation scheme that can, using user directives, interleave the individual sequential programs with fine granularity without causing any interference.

2 Seuss Programming Model and Notation

We describe a programming model and a publication notation. The notation is intended for implementation on top of a variety of host languages. Therefore, no commitment has been made to the syntax of any particular language. The notation is fully described in [13].

The notation is described using BNF. All non-terminal identifiers are in Roman and all terminal identifiers are in boldface type. The special symbols used as meta symbols of the BNF are

::= { } [ ] ∨ ( )

and the special symbols used as terminals are

| / ; : ::

A syntactic unit enclosed within "{" and "}" may be instantiated zero or more times. A syntactic unit enclosed within "[" and "]" may be instantiated zero or one time. The symbol ∨ is used for alternation; in the right hand side of a production, $(p \vee q)$ denotes that a choice is to be made between the syntactic units p and q in instantiating this production. We omit the parentheses, (and), when no confusion can arise.

2.1 Program Structure

The basic constructs of a Seuss program are *box*, *clone* and *procedure*. A box defines a type (or *class* in object-oriented programming), a clone is an instance of a box, and procedures (actions and methods) are the constituents of boxes. A box has a local state and it includes procedures by which its local state can be accessed and updated. Procedures in a box or a clone may call upon procedures of other clones. Boxes are used to encode processes as well as the communication protocols for process interactions; therefore, it is necessary only to develop the methodology for programming and understanding boxes and their component procedures.

box ::= **box** box-name {component} **end**
component ::= box ∨ clone ∨ variable ∨ procedure
box-name ::= identifier

A box may contain declarations of boxes; therefore boxes may be nested to arbitrary depth. The hierarchical structure of a program can be described by a *syntax tree*: For box A that has a component box (or clone) B, A is the $parent$ of B and B a $child$ of A.

Program A *program* consists of a box – that describes the program code – and a clone – that instantiates the box (typically, we combine the box and the clone, as described in section 3).

Library There is a special box, $Library$, that includes the ubiquitous programming constructs such as channels and semaphores as boxes. By convention, every user program is a component of $Library$ and $Library$ is the root of the syntax tree that describes a program. Consequently, every box and clone in a user program has a parent. Any component box of $Library$ may be instantiated within the user program.

Scope Rules The scope rules determine the names – of boxes, clones, variables and procedures – that can be referenced within a box. The rules are similar to those in other block-structured languages, such as PASCAL; see Chapter 2 of [13] for details.

2.2 clone

clone ::= **clone** clone-name: box-name [**init** clone-initialization]
clone-name ::= identifier

A clone is an instance of a box. The form of a clone declaration is

clone $c : B$ **init** $q_0; q_1, ...$

The initialization is optional; in the case shown above, $q_0, q_1, ...$ are total-methods of B, and they are executed sequentially when c is instantiated.

2.3 variable

variable ::= **var** variable-name: type [**init** variable-initialization]
variable-name ::= identifier

We rely on traditional notations for declarations and initializations of variables.

2.4 procedure

We propose two distinct kinds of procedures, to model terminating and potentially non-terminating computations – representing computations of wait-free programs and multiprograms, respectively. The former can be assigned a semantic with pre- and post-conditions, i.e., based on its possible inputs and corresponding outputs *without* considerations of interference with its environment. Multiprograms, however, cannot be given a pre- and post-condition semantic because on-going interaction with the environment is of the essence. We distinguish between these two types of computations by using two different kinds of procedures: a *total* procedure never waits (for an unbounded amount of time) to interact with its environment whereas a *partial* procedure may wait, possibly forever, for such interactions. In this view, a P operation on a semaphore is a partial procedure – because it may never terminate – whereas a V operation is a total procedure. A total procedure models wait-free, or *transformational*, aspects of programming and a partial procedure models concurrent, or *reactive*, aspects of programming[10]. Our programming model does not include *waiting* as a fundamental concept; therefore, a (partial) procedure does not wait, but it *rejects* the call, thus preserving the program state.

procedure ::= partial-procedure $\vee$ total-procedure
partial-procedure ::= **partial** partial-method $\vee$ partial-action
total-procedure ::= **total** total-method $\vee$ total-action
partial-method ::= **method** head :: partial-body
partial-action ::= **action** [head] :: partial-body
total-method ::= **method** head :: total-body
total-action ::= **action** [head] :: total-body

A procedure is either **partial** or **total**; also, a procedure is either a **method** or an **action**. Thus, there are four possible headings identifying each procedure. Each procedure has a head and a body; the head is optional for actions. The procedure head is similar to the form used in typical imperative languages; it has a procedure name followed by a list of formal parameters and their types. The procedure body is different for partial and total procedures.

partial-body

partial-body ::= alternative $\{$(| alternative) $\vee$ ($\not|$ alternative)$\}$
alternative ::= precondition [; preprocedure] $\rightarrow$ total-body

The body of a partial procedure consists of one or more alternatives. Each alternative is *positive* or *negative*: the first alternative is positive; an alternative preceded by | is positive and one preceded by $\not|$ is negative. Each alternative has a precondition, an optional preprocedure and a body that is a total-body. A precondition is a predicate; it is constrained to name only the variables of the box in which the procedure appears. The precondition of at most one alternative of a partial procedure holds in any state, i.e., the preconditions are pairwise disjoint. A preprocedure is the name of a partial method in a visible clone; see scope rules in section 2.1.

total-body A total-body is a wait-free program; in this document, we use only sequential programs for total-body. The body may call upon total-methods of other clones in its scope. No partial-method may be called from a total-body. The total-body can contain all the constructs of typical sequential programs including local declarations (variables, procedures, functions, etc.). It should be established by the programmer that every execution of a total-body terminates, perhaps in a *failed* state.

Procedure Execution A method is executed when it is called. A total-method always *accepts* calls; its body is executed whenever it is called. A partial-method *accepts* or *rejects* each call; it accepts a call if and only if one of its positive alternatives accepts the call, and it rejects the call otherwise. An alternative, positive or negative, accepts a call in a given state as follows. An alternative of the form $p \rightarrow S$ accepts the call if p holds; then its body, S, is executed and control is returned to its caller. An alternative of the form $p;\ h \rightarrow S$ accepts a call provided p holds and h accepts the call made by this procedure (using the same rules, since h is also a partial procedure); upon completion of the execution of h the body S is executed, and control is returned to the caller. Thus, an alternative rejects a call if the precondition does not hold, or if the preprocedure, provided it is present, rejects the call. Note that, since the precondition of at most one alternative of a partial procedure holds in a given state, at most one alternative will accept a call (if no alternative accepts the call, the call is rejected). It follows that the state of the caller's clone is unchanged whenever a call is rejected, though the state of the called clone may be changed because a negative alternative may have accepted the call.

The execution of an action is similar to that of a method, even though the former is not called. A total action always accepts and executes its body. An execution of a partial action is accepted if any one of its positive alternatives accepts, else the execution is rejected.

3 Program Execution

A *tight* execution of a program is an infinite sequence of steps where each step consists of executing an action of a clone. The choice of actions is arbitrary except that each action of each clone appears in an infinite number of steps of a tight execution. We specify below the run-time structure of the program; it is a tree of clones and the actions of the program is the set of all actions at the nodes of the tree.

A program consists of a box followed by a clone that is an instantiation of the box. During the execution of a program only the instantiated clones exist; all the boxes have been eliminated. We employ an *instantiation tree* to display the run-time structure of a program. We start by describing the rules for instantiating a clone of the form

clone $c : B$ **init** $q_0; q_1, ...$

Instantiation tree of c has a root labeled c and it consists of the instances of the variables, methods and actions of B. The clones of B are recursively instantiated and the resulting trees become the subtrees of c. The variables of c are initialized as specified in

their declarations in the **var** command, and then by executing the sequence of methods $q_0; q_1, ...$ Observe that the box declarations have been eliminated; in particular, $Library$ does not appear and the root of the instantiation tree is the clone corresponding to the user-program.

We require that the programmer establish a partial order $lower$ among the clones of the instantiation tree such that a procedure at node y calls a method at node x only if $x\ lower\ y$. A simple syntax-based scheme is to ensure that the definition of a clone lexically-precedes any reference to it.

Notational Convention It is often the case that a box has a single clone, as shown below.

box B
  body of B
end

clone $c : B$ **init** $q_0; q_1, ...$

In this case, we eliminate the explicit box declaration, and the introduction of symbol B by using the convention that the program fragment given below stands for the one above.

clone c
  body of B
end init $q_0; q_1, ...$

Normally, a program is written in this manner, as a single clone.

4 An Example

This example, attributed to Hamming, is taken from Dijkstra[6]. The purpose of the example is not to show a clever algorithm, but to illustrate various features of Seuss.

It is required to compute the sequence of integers of the form $2^i \times 3^j \times 5^k$ in increasing order, for all natural numbers i, j, k. Our solution follows the treatment in Section 8.2 (page 182) of Chandy and Misra[4], and it is sketched briefly below.

The program $Hamming$ (a clone) consists of the boxes, $mult$, $merge$, and $produce$. Boxes $mult$ and $merge$ are $Fifo$ channels, where $Fifo$ is declared in $Library$. $Hamming$ contains a single partial method, $next$, that yields the next number, g, in the desired sequence. Since, $2 \times g, 3 \times g, 5 \times g$ are also numbers of the desired sequence, $next$ sends these numbers along the $Fifo$ channels $mult[1]$, $mult[2]$, $mult[3]$, respectively. The role of $produce$ is to merge the increasing sequences received along $mult[1]$, $mult[2]$, $mult[3]$, and send the resulting increasing sequence along $merge$. Merging is done by using three variables, h[1..3], where $h[i]$ is the last number received along $mult[i]$; if all numbers received have already been output then $h[i]$ is 0. Box $produce$ has two partial actions, $read$ and $write$. The i^{th} procedure within $read$, $1 \leq i \leq 3$, is dedicated to receiving the next value from $mult[i]$ provided $h[i] = 0$. Procedure $write$ outputs the smallest of the h values, when they are all nonzero, along $merge$ and updates h. The computation is started by having the value 1 in the channel $merge$, initially.

```
clone Hamming
  clone mult[1..3]: Fifo
  clone merge: Fifo init put(1)

  clone produce
    var h[1..3]:nat init 0, f: nat
    partial action read::
      (‖i : 1 ≤ i ≤ 3 :  h[i] = 0;  mult[i].get(h[i]) →  skip )

    partial action write::
      h[1] ≠ 0 ∧  h[2] ≠ 0 ∧  h[3] ≠ 0 →
        f := min(h[1], h[2], h[3]);  merge.put(f);
        if f = h[1] then h[1] := 0;
        if f = h[2] then h[2] := 0;
        if f = h[3] then h[3] := 0
  end {produce}

  partial method next(g: nat)::
    ; merge.get(g) →
      mult[1].put(2 × g);
      mult[2].put(3 × g);
      mult[3].put(5 × g)
end {Hamming}
```

5 Related Work

Our work incorporates ideas from serializability and atomicity in databases[1], notions of objects and inheritance[11], Communicating Sequential Processes[7], i/o automata[9], and Temporal Logic of Actions[8]. A partial procedure is similar to a database (nested) transaction that may commit or abort; the procedure commits (to execute) if its precondition holds and its preprocedure commits, and it aborts otherwise. A typical abort of a database transaction requires a rollback to a valid state. In Seuss, a partial procedure does not change the program state until it commits, and therefore, there is no need for a rollback. The form of a partial procedure is inspired by Communicating Sequential Processes[7]. Our model may be viewed as a special case of CSP because we disallow nested partial procedures.

Seuss is an outgrowth of our earlier work on UNITY [4]. A UNITY program consists of statements each of which may change the program state. A program execution starts in a specified initial state. Statements of the program are chosen for execution in a non-deterministic fashion, subject only to the (fairness) rule that each statement be chosen eventually. The UNITY statements were particularly simple – assignments to program variables – and the model allowed few programming abstractions besides asynchronous compositions of programs. Seuss is an attempt to build a compositional model of multiprogramming, retaining some of the advantages of UNITY. An action is similar to a statement, though we expect actions to be much larger in size. We have

added more structure to UNITY, by distinguishing between total and partial procedures, and imposing a hierarchy over the boxes. Executing actions as indivisible units would extract a heavy penalty in performance; therefore, we have developed the theory that permits interleaved executions of the actions. Programs in UNITY interact by operating on a shared data space; Seuss boxes, however, have no shared data and they interact through procedure calls only. In a sense, boxes may only share boxes. As in UNITY, the issues of deadlock, starvation, progress (liveness), etc., can be treated by making assertions about the sequence of states in every execution. Also, as in UNITY, program termination is not a basic concept. A program has reached a *fixed point* when preconditions of all actions are *false* ; further execution of the program does not change its state then, and an implementation may terminate a program execution that reaches a fixed point. We have developed a simple logic for UNITY (for some recent developments, see [15], [14], [3]) that is applicable to Seuss as well.

References

1. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.
2. J.C. Browne et al. A language for specification and programming of reconfigurable parallel computation structures. In *Int. Conf. of Parallel Processing, Bellaire, Michigan*, pages 142–149. IEEE, Aug 1982.
3. K. M. Chandy and B. A. Sanders. Towards Compositional Specifications for Parallel Programs. In *DIMACS Workshop on Specifications of Parallel Algorithms*, Princeton, NJ, May 9-11 1994.
4. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.
5. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
6. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
7. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, London, 1984.
8. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
9. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989.
10. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1991.
11. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, NJ, 1997.
12. R. Milner. *Communication and Concurrency*. International Series in Computer Science, C. A. R. Hoare, Series Editor. Prentice-Hall International, London, 1989.
13. Jayadev Misra. *A Discipline of Multiprogramming*. Unpublished Manuscript Available at the following URL, ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z.
14. Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.
15. Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.