Seuss: What the Doctor Ordered

Lorenzo Alvisi Rajeev Joshi Calvin Lin Jayadev Misra

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

Reconciling the conflicting goals of simplicity *and* efficiency *has traditionally been a major challenge in the development of concurrent programs.* **Seuss** *[16] is a methodology for concurrent programming that attempts to achieve the right balance between these competing concerns. The goal of Seuss is to permit a disentanglement of the issues of correctness and efficiency. On the one hand, programmers can reason about Seuss programs by assuming a single thread of control; on the other hand, implementation designers can exploit design knowledge in achieving better performance. This paper provides a short overview of the Seuss programming model and describes the main challenges in designing an efficient implementation of Seuss and in applying Seuss to large applications.*

1. Introduction

Distributed programming has long been recognized as being far more difficult than sequential programming. The myriad of interactions that may occur in a distributed system introduces a complexity that is far beyond the reach of the methodologies used for sequential designs. Consequently, most research in distributed programming has been directed at managing this complexity by limiting the interactions that may occur in a system, and a variety of synchronization mechanisms have been proposed, e.g., semaphores, monitors, barriers, remote procedures, and atomic transactions. These mechanisms typically differ in the granularity of interactions they restrict; reducing interactions to large granularities usually results in programs that are simpler to understand but less efficient to execute. One of the main challenges has been the development of a programming methodology that allows programmers to retain intellectual control over their programs without having to sacrifice efficiency in implementation.

Seuss [16] represents a departure from traditional approaches by making a fundamental distinction between the concurrent and the sequential aspects of distributed programming: We believe that concurrent programs are best structured as large sequential components that interact only through small but intricate sections of code. Program design in Seuss involves separately considering (i) the programming of each component and (ii) the orchestration of the executions of these components. We believe that different theories and methodologies are appropriate for these two tasks. For instance, (i) often involves simple sequential programming techniques, whereas (ii) usually involves consideration of component interactions and is often largely independent of the low-level details of the design.

The goal of Seuss is to disentangle the concerns of correctness and efficiency. Programmers reason about Seuss programs as if they were executed with a single thread of control; this simplifies correctness proofs. At the same time, implementation designers can exploit design knowledge in order to execute programs using multiple threads; this leads to better performance.

The theoretical aspects of Seuss have been described elsewhere [17], [16]. In this paper, we provide a short overview of Seuss and identify the main challenges in applying Seuss to large distributed applications and in developing efficient implementations of Seuss programs.

2. The Programming Model

The programming model of Seuss is simple. A Seuss program is a collection of *box* and *clone* declarations. A box consists of variables, which define local state, and procedures, which update and access this state. A clone is an instance of a box. Informally speaking, a box is like a class (in object-oriented terminology) and a clone is like a class instance. There are, however, some key differences between our notion of box and the concept of a class. For instance, boxes are required to have a certain structure, and the call graph of a Seuss program is required to be acyclic. We have chosen a neutral terminology to avoid confusion with these traditional concepts.

Clones are the only unit of sharing in Seuss. In particular, there are no shared variables in Seuss programs, and data is exchanged only through procedure calls on clones. This leads to a uniform programming model in which there is no distinction between computation and communication. For instance, we would model traditional FIFO message communication by declaring a clone with a local sequence variable (for storing messages) and two procedures 'put' and 'get' for accessing this sequence. This clone could then be used by other clones in order to exchange data.

Each procedure in a box is either a *method* or an *action*. Both methods and actions may change local state and make calls upon methods of other clones (subject to certain restrictions, described in the sections below). The difference is that methods are invoked only when called by procedures of other clones, whereas actions are executed autonomously during program execution. Decisions of when to execute each action are left to the implementation designer – different implementations may choose to execute actions using different schedules, provided they satisfy the following fairness constraint: every action (from every clone) is executed infinitely often.

Seuss provides programmers with limited control over the scheduling of actions. For instance, the sequencing of two actions has to be programmed explicitly. We feel that this loss of flexibility is only to be expected when providing high-level abstractions. As an analogy, we note that machine language offers complete control over sequential program execution: instructions may be treated as data, data types may be ignored entirely, and control flow may be altered arbitrarily. Such flexibility is appropriate only when programs are very short, and this flexibility is abandoned in structured programming languages in favor of readability, correctness, modularity, and portability. We believe that concurrent programming requires even stronger restrictions if programs are to remain intellectually manageable.

The execution of a Seuss program is conceptually very simple: every action is executed infinitely often. We place restrictions on program structure so that each action terminates and can be understood in isolation, without considering complex interactions with other actions. This allows programmers to reason about Seuss programs by assuming that actions are executed one at a time, in a non-interleaved fashion. In particular, this means that there is no notion of *waiting* in Seuss; instead, this notion is replaced by that of *rejection* (see Section 2.3 below). As an example, consider the P operation on a semaphore. Traditionally, when attempted in a state in which the semaphore is unavailable, the operation causes the caller to wait. In Seuss, the call to the P operation is rejected if the semaphore is not available and accepted otherwise. To acquire the semaphore, a caller repeatedly attempts the P operation until it is accepted.

2.1. Procedures

Our theory makes a fundamental distinction between two kinds of computations, viz., (i) terminating, or *total*, procedures and (ii) potentially nonterminating, or *partial*, procedures. Traditional sequential techniques – e.g., pre- and post-conditions – may be used to reason about a total procedure, whereas reasoning about a partial procedure involves some consideration of interaction with other procedures. The next two sections describe total and partial procedures in greater detail.

2.2. Total procedures

A total procedure can be assigned a meaning based only on its inputs and its outputs. Examples of total procedures include sorting a list, finding a minimum spanning tree in a graph, and sending a job to an unbounded print queue. A total procedure need not be deterministic – e.g., *any* minimum spanning tree could be returned by a procedure that computes a minimum spanning tree. Furthermore, a total procedure need not be implemented on a single processor. A total procedure may even be a parallel program that admits asynchronous execution, provided that it is guaranteed to terminate and that its effects may be understood only through its inputs and its outputs.

Total procedures may call only total procedures. When a total procedure is called it may either terminate normally or terminate in a failed state. A failure is caused by a programming error and occurs when a procedure is invoked in a state in which it is not defined, e.g., an attempt to divide by zero.

2.3. Partial procedures

Unlike a total procedure, a partial procedure cannot be assigned a meaning based only on its inputs and its outputs. Execution of a partial procedure typically involves interaction with the environment, which may not be ready for the interaction. Examples of partial procedures include acquiring a semaphore, getting a message from a channel, and sending a job to a bounded print queue. Traditionally, such interactions would cause processes to wait, and would involve consideration of the behavior of other processes. However, there is no notion of waiting in Seuss. Instead, as described below, we impose some restrictions on the structure of partial procedures; these restrictions allow us to understand partial procedures in isolation.

In this paper we restrict ourselves to the following simple form of partial procedures (the general case is described elsewhere [16]):

$$p \,;\, h \;\rightarrow\; S$$

where p is the *precondition*, h is the *preprocedure* and S is the *body* of the procedure. The precondition is a predicate on the state of the clone in which the partial procedure is declared. The preprocedure, which is optional, is a call on a partial procedure in another clone. The body S is required to terminate, though it may include calls on total procedures of other clones. Note that the body does not contain any calls to partial procedures, and there is at most one call in the preprocedure to another partial procedure. (This condition ensures that the execution of a partial procedure always terminates [16].)

A partial procedure responds to each call made upon it by *accepting* or *rejecting* the call. A partial procedure of the form $p \rightarrow S$, where the preprocedure is absent, responds to a call as follows. If p holds, S is executed and the call is accepted; otherwise, S is not executed and the call is rejected. A partial procedure g of the form $p\ ;\ h \rightarrow S$ responds to a call as follows. If p holds, then h is attempted; if h accepts, then g executes S and accepts. If p does not hold, or if h rejects, then g rejects. (In Section 2.4 we will describe additional restrictions on program structure that ensure that this discipline is well-founded.)

As an illustration of the difference between total and partial procedures, consider the P and V operations of a binary semaphore. If the semaphore value is 0, then the application of V changes it to 1. But if the semaphore value is 1 prior to the application of V, there are at least four possibilities for a traditional implementation: (i) the operation is implemented as a $skip$ (i.e., the semaphore value remains 1 and the operation terminates), (ii) the operation fails, i.e., it changes the semaphore value arbitrarily either to 0 or to 1, (iii) the operation waits for the semaphore value to become 0, and (iv) the operation never terminates. If we adopt implementations (i) or (ii) then we may regard the V operation as a total procedure. With implementation (iii) the operation is viewed as a partial procedure. We view implementation (iv) to be a programming error. By contrast, the P operation is a partial procedure, which accepts only when the value of the semaphore is 1. A partial procedure that calls upon P could be of the form

$$true\ ;\ P \rightarrow S \quad .$$

To simulate waiting, the execution of this procedure is attempted repeatedly as long as P rejects.

2.4. Programs

A program is a finite set of clones, ordered by the 'calls' relation on the procedures. The call graph of a Seuss program is required to be acyclic; this results in an irreflexive partial ordering on the clones of a program. A procedure in a clone may call procedures in clones that are lower in the partial order. This restriction implies that a partial procedure at a lowest level is of the form $p \rightarrow S$, where the preprocedure is absent and the body S contains no procedure calls. A total procedure at a lowest level contains no procedure calls.

We have formally described the model and the notation elsewhere and have shown how boxes and clones can be combined to form programs [16].

2.5. Tight Executions

Executions of Seuss programs are classified into *tight* and *loose* executions. A tight execution is an infinite sequence of steps. In each step, some action of some clone is chosen and executed. Actions may be chosen in any order, as long as every action is chosen infinitely often. A tight execution is easy to reason about because the execution of each action completes before another action begins. Given the semantics of the procedures that it calls, the execution of each action can be understood from its text alone, without consideration of interference by other procedures.

2.6. Loose Executions

Tight executions are easily implemented on uniprocessors. For implementation on multiprocessors, however, it should be possible to execute non-interfering actions concurrently. Our notion of non-interference is similar to the notion of 'commutativity' in database theory, where the executions of commuting transactions may be interleaved without introducing inconsistencies. The analogue of commutativity in our theory is called *compatibility*, which, like commutativity, is a relation between actions. Compatibility is, however, a weaker notion than traditional commutativity; thus it admits greater concurrency. A formal definition of compatibility, and further discussion of this topic, can be found elsewhere [17].

2.7. Examples

In this section, we show two small examples of Seuss programs, which illustrate the concepts described above.

Message communication

An unbounded FIFO channel is designed as a box with a local variable of type sequence and two methods. Total method put appends an item to the end of the message sequence and partial method get removes and returns the first item in the message sequence, provided that the sequence is nonempty. We define a polymorphic version of the channel where the message type is arbitrary. In the put method we use a colon (:) in the assignment to denote concatenation.

box $FifoChannel$ **of** type
  var r : **seq of** type **init** $\langle\rangle$
    { r is initially empty }
  partial method $get(x$: type) ::
    $r \neq \langle\rangle \ \rightarrow \ x, r := r.head, r.tail$
  total method $put(x$: type) :: $r := r : x$
end $\{FifoChannel$ of type $\}$

Task Dispatcher

Next, we show a task dispatcher that is interposed between a set of clients and a set of servers. A client generates a sequence of tasks, where each task has a priority between 0 and N. A server attempts to process a task whenever it is idle. However, a server can process tasks below a certain priority only; this priority is passed as the parameter p to the procedure get shown below.

It is easy to see that a task dispatcher is nothing but a glorified channel; it has two methods, put and get. A client calls put, with a task and priority as parameters, to deposit a task in the channel. A server calls get, with a parameter p, to obtain a task with the highest priority at or below p, if such a task exists. In the following solution, $r[i]$ is a queue of tasks of priority i that have been deposited by the clients but not yet been processed by the servers.

box $dispatcher$
  var $r[0..N]$: **seq of** task **init** $\langle\rangle, i : 0..N$
  partial method $get(x$: task, $p : 0..N)$ ::
    {get a task of priority p or lower,
    as close to p as possible}
    $(\exists\, j :: 0 \leq j \leq p \wedge r[j] \neq \langle\rangle) \ \rightarrow$
      $i := p$;
      while $r[i] = \langle\rangle$ **do** $i := i - 1$ **enddo** ;
      $x, r[i] := r[i].head, r[i].tail$
  total method $put(x$: task, $p : 0..N)$ ::
    $r[p] := r[p] : x$
end $\{dispatcher\}$

Note that this solution does not guarantee that every task will eventually be removed. (The above design can, however, be modified to achieve starvation-freedom.)

3. Related Work

Seuss is an outgrowth of earlier work on UNITY [5]. A UNITY program contains a set of declarations, which define the state space, and a set of statements, each of which may change the program state. A program execution starts in any of a set of specified initial states. Statements of the program are chosen for execution in a nondeterministic fashion, subject only to the unconditional fairness rule that each statement be chosen infinitely often. Statements in UNITY are particularly simple – assignments to program variables – and the model allows few programming abstractions besides asynchronous compositions of programs. Programs interact solely through operations on a set of shared variables.

Seuss is an effort at building a compositional model of concurrent programming, while retaining some of the advantages of the simplicity of UNITY. An action is similar to a UNITY statement, although we expect actions to be much larger in size. Seuss has more structure than UNITY in that it distinguishes between total and partial procedures and imposes an ordering on clones. Executing actions as indivisible units would exact a heavy penalty in performance; therefore, we have developed a theory that permits interleaved executions of action bodies. Unlike UNITY, subprograms in Seuss have no shared data, and they interact through procedure calls only. As in UNITY, the issues of deadlock, starvation, progress (liveness), etc., are 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 the 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 that is currently being extended to Seuss [19, 18, 6, 1].

Seuss also incorporates ideas from serializability theory and atomic transactions in databases [4], object-oriented programming [14], Communicating Sequential Processes (CSP) [9], i/o automata [13, 12], and the Temporal Logic of Actions (TLA) [11]. A partial procedure is similar to a database transaction that may commit or abort; the procedure commits to execute if its precondition holds and its preprocedure commits, and it aborts (i.e., rejects) 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 Hoare's work on CSP [9]. Our model may be viewed as a special case of CSP because we disallow nested partial procedures. Lynch and Tuttle, in their work on i/o automata, have explored similar issues; however, they do not advocate a distinction between partial and total procedures, as we do in Seuss. The design of our logic has been influenced by Lamport's work on TLA [11], and by the action sequences developed by Milner in CCS [15]; Seuss admits both state-based as well as action-based reasoning and we intend to exploit both forms.

4. Design Issues

We have designed and implemented a simple prototype for Seuss on a network of SUN Sparcstations. The sequential programs in total procedure bodies are written in C++. A compiler [10] generates C++ code from Seuss programs, introducing calls to the PVM messaging library [8] wherever communication between clones is required. Each clone is mapped to a separate process, and each procedure call results in an exchange of messages between processes. Actions are scheduled by a single *scheduler*, which is also a process in the system. This scheduler is designed so that (i) only compatible actions execute concurrently, and (ii) every action is executed infinitely often. We have developed an algorithm for the scheduler that is fully nondeterministic in the following sense: given a set of actions, along with a compatibility relation on the set, the algorithm is capable of generating *any* schedule that satisfies the two restrictions listed above.

The implementation was intended as a proof of concept and as a means of identifying the problems that need to be addressed in order to obtain reasonable efficiency. The prototype is inadequate in several ways:

- It provides no way for the programmer to control locality, which can be critical to achieving good performance.
- It assigns each clone to a separate process, so communication between clones can be prohibitively expensive.
- The scheduler is implemented as a single process, which becomes a bottleneck when the number of processes is large.

We are currently working on an improved implementation which will address these issues. The remainder of this section outlines the approach that we plan to follow.

User-Supplied Directives. To provide locality information and performance-critical information to the system, we will allow programmers to add directives to their Seuss programs. These directives will not affect program correctness. Instead, they will act as hints, which will aid the implementation in generating efficient code. This approach preserves Seuss's philosophy of separating concerns: programmers may reason about correctness independently of the presence of the hints, while sophisticated implementations may still be able to exploit design knowledge in achieving better performance.

One such directive will describe memory requirements, which are fundamental to performance when clones encapsulate large amounts of data. In traditional approaches, this information is gathered statically by the compiler; however, Seuss programs may consist of total procedures written in several languages – possibly even machine code – and a compiler will often be unable to determine such information from program text alone.

Another type of directive will describe the size of parameters to procedures. Since all interactions in Seuss occur via procedure calls, these directives will describe the granularity of communication, which might help the compiler in mapping clones to processes.

A third type of directive will describe the environment in which the program will execute by providing information such as the types of the processors available in the system, their relative speeds, their available memory, the network topology, and bandwidth characterizations. Such information may be useful in determining a good mapping in a heterogeneous environment.

Multi-threaded Execution. To address the second inadequacy, we will use multi-threaded execution with multiple clones residing in each process. A key issue will be the mapping of clones to processes in order to maximize concurrency, maximize data locality, and minimize interprocess communication. Programmer-supplied directives will be used to improve the compiler's ability to map clones and to improve the runtime system's ability to schedule them.

Distributed Scheduler. The third fundamental change to the prototype is to distribute the scheduler. Here, the fundamental tradeoff is between gathering the information needed to produce good schedules and minimizing the overhead of the scheduler. We plan to conduct experiments to assess the performance impact of good scheduling, after which we will be able to identify effective heuristics for choosing the proper level of communication.

Code Reuse. To facilitate code reuse we will extend Seuss to allow total procedures to be written in sequential languages such as C and Fortran. Note that the properties of the Seuss model – viz., that correctness depends only on the interactions among boxes – are ideally suited to achieving this goal.

Seuss-Specific Optimizations. We have identified a number of Seuss-specific optimizations. Our scheduler guarantees unconditional fairness, which states that each action is executed infinitely often. Within this constraint, there is wide latitude in choosing which actions to execute. It's clearly beneficial to choose actions whose preconditions are true and whose preprocedures reject, as opposed to actions which result in spin waiting. In some situations, it may also be beneficial to cache the state of preconditions. Where possible, the compiler will determine when these cached values may change, and scheduling decisions will be made based on these cached values. Again, directives may play a rôle in helping the compiler determine how one procedure may affect the guard of another.

Ideally, if the preconditions of several partial actions evaluate to true, it would be preferable to attempt calls on all their preprocedures and execute the action whose call is accepted first. Unfortunately, this strategy does not work in general, because the execution of a preprocedure of one action may falsify the precondition of another action. If the preprocedure of the latter has caused a state change, then the assumption of a single thread of control may no longer be applicable. There is, however, a special case in which such an optimization works, viz., when the precondition of a partial procedure is not falsified by execution of other procedures in the same clone. In such situations, it is possible to overlap a call to the preprocedure with the execution of another procedure of the same clone.

Communication Optimizations. The communication among the set of clones in a program may result in the exchange of many small messages if the clones execute on different processors. One way to minimize the overhead of the messaging required for these calls is to combine multiple requests into a single message. This is possible when multiple requests (from multiple clones in a single process) are destined to the same processor. The receiving process is then responsible for extracting the set of calls from the messages it receives. Such optimizations need great care in order to ensure that they do not adversely affect scheduling or introduce the possibility of deadlock.

5. Applications

One goal of Seuss is to simplify the design of complex distributed programs. We are currently applying Seuss to the following domains.

Cache Coherence Protocols. With the performance advantage that application-specific protocols can provide [7], there has been a new demand for designing customized protocols. However, cache coherence protocols are notoriously difficult to debug and serve as prime examples of distributed applications in which all of the complexity is concentrated in small sections of code. We have expressed several caching algorithms in Seuss, with the goal of proving their correctness. These include the lazy caching algorithm [2], the Sprite cache consistency protocol [20], and the consistency algorithm used in XFS [3].

Design of Manufacturing Control. One particularly promising application appears to be the development of manufacturing controllers. These controllers are configured in a hierarchical manner and are currently designed in a bottom-up manner. While at the lowest level each controller can be optimized individually to improve efficiency, this design approach makes it difficult, if not impossible, to evaluate and tune the controllers with respect to global performance. With Seuss we can provide a methodology for top-down control, which should make it possible to prove and optimize controllers at all levels of the hierarchy. We expect that this will result in better control of the manufacturing process with respect to efficiency, cost, and storage requirements.

6. Conclusion

This paper has provided an overview of Seuss, an ongoing project aimed at simplifying the construction of complex distributed applications. We have presented the Seuss programming model and identified some key issues that we are addressing as we attempt to build an efficient Seuss implementation and apply Seuss to large applications. We are currently working on the correctness proofs of existing cache consistency algorithms [2], [20], [3]. We also plan to begin work soon on a more efficient implementation that incorporates some of the optimizations described in Section 4.

References

[1] W. E. Adams. Concurrent programming with a single thread of control. *Dissertation Proposal, University of Texas, Department of Computer Science*, 1995.

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, Jan. 1993.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, Feb. 1996.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.

[5] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[6] 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.

[7] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[9] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, London, 1984.

[10] I. H. Krüger. An experiment in compiler design for a concurrent object-based programming language. Master's thesis, University of Texas, Department of Computer Science, 1996.

[11] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[12] N. Lynch, M. Merritt, W. Weihl, and aln Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[13] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989.

[14] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall international, London, 1988.

[15] R. Milner. *Communication and Concurrency*. International Series in Computer Science, C. A. R. Hoare, Series Editor. Prentice-Hall International, London, 1989.

[16] J. Misra. A discipline of multiprogramming. Work in progress, available online at ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z .

[17] J. Misra. A discipline of multiprogramming. In G. E. Blelloch, K. M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, volume 18, pages 357–381, Providence, RI, 1994. DIMACS (Series in Discrete Mathematics), American Mathematical Society.

[18] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[19] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[20] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, Feb. 1988.