A Reduction Theorem for Concurrent Object-Oriented Programs

JAYADEV MISRA[1]

ABSTRACT A typical execution of a concurrent program is an interleaving of the threads of its components. It is well known that the net effect of a concurrent execution may be quite different from the serial executions of its components. In this paper we introduce a programming notation for concurrent object-oriented programs, called **Seuss**, and show that concurrent executions of its programs are, under certain conditions, equivalent to serial executions. This allows us to reason about a Seuss program as if its components will be executed serially whereas an implementation may execute its components concurrently, for performance reasons.

1 Introduction

A typical execution of a concurrent program is an interleaving of the threads of its component programs. For instance, consider a concurrent program that has α and β as component programs, where the structures of α, β are as follows:
α:: α_1; α_2; α_3, and
β:: β_1; β_2; β_3.
The concurrent execution $\alpha_1\ \beta_1\ \alpha_2\ \beta_2\ \alpha_3\ \beta_3$ interleaves the two sequential executions. It is well known that the net effect of a concurrent execution may be quite different from the serial executions of the components. In this example, suppose α_1, β_1 are "read the value of variable x", α_2, β_2 are "increment the value read", and α_3, β_3 are "store the incremented value in x". Then, the given interleaved execution increases the value of x by 1 whereas an execution in which the threads are not interleaved increases x by 2.

The method of reduction was proposed by Lipton[3] to simplify reasoning about concurrent executions. Lipton develops certain conditions under which the steps of a component program may be considered indivisible (i.e., occurring sequentially) in a concurrent execution. A step f in a component

[1]This material is based in part upon work supported by the National Science Foundation Award CCR–9803842.

is a *right-mover* if for any step h of another component whenever fh is defined then so is hf and they yield the same result (i.e., their executions result in the same final state). Similarly, g is a *left-mover* if for any h of another component hg is defined implies gh is defined, and $hg = gh$. Lipton shows that a sequence of steps of a component, $r_0\ r_1\ ...\ r_n\ c\ l_0\ l_1\ ...\ l_m$, may be considered indivisible for proof of termination of a concurrent program if each r_i is a right-mover, l_j a left-mover and c is unconstrained. This result has been extended to proofs of more general properties by Lamport and Schneider [2], Misra [4], and, more recently, by Cohen and Lamport [1].

In section 2, we introduce a programming notation for concurrent object-oriented programming, called **Seuss**. Briefly, a seuss program consists of **boxes**; a box is similar to an object instance. A box has local variables whose values define the state of the box. A box has **actions** and **methods**, both of which will be referred to as **procedures**. Actions are executed autonomously; a method is executed by being called by an action or a method of another box. In section 2.2, we introduce two different execution styles for programs, *tight and loose*. In a tight execution an action is completed before another action is started. In a loose execution the actions may be executed concurrently provided they satisfy certain *compatibility* requirements. A tight execution, being a single thread of control, may be understood more easily than a loose execution. Loose execution, on the other hand, is the norm where the computing platform consists of a large number of processors.

In this paper we develop a reduction theorem that establishes that for every loose execution there is a corresponding tight execution: if a loose execution of some finite set of actions starting in state s terminates in state t then there is a tight execution of those actions that can also end in state t starting in state s. This result is demonstrated by prescribing how to transform a loose execution into a tight execution in the above sense. This correspondence allows a programmer to understand a program in terms of its tight executions —a single thread of control— whereas an implementation may exploit the available concurrency through a loose execution.

The proof of the reduction theorem is considerably more difficult in our case because (1) procedure calls introduce interleavings of "execution trees" rather than execution sequences, and (2) executions of any pair of actions may be interleaved provided the actions are compatible. The notion of compatibility is central to our theory. Roughly, two procedures are compatible if their interleaved execution may be simulated by executing them one after the other in some order. We give an exact definition and show how compatibility of procedures may be proven.

Compatibility information can not be deduced automatically. Yet it is unrealistic to expect the user to provide this information for all pairs of procedures; in most cases, different boxes will be coded by different users, and no user may even know which other procedures will be executing. Therefore, we have developed a theory whereby compatibility of procedures belonging

to different boxes may be deduced automatically from the compatibility information about procedures belonging to the same box. Users simply specify which procedures in a box are compatible and an algorithm then determines which pairs of actions are compatible, and may be executed concurrently.

Plan of the paper

In the next section, a brief introduction to Seuss is given; the reader may consult [5] for a detailed treatment. An abstract model of Seuss is given in section 3. In section 2.1 we state certain restrictions on programs which we elaborate in section 4. The definition of compatibility appears in section 5. A statement of the reduction theorem and its proof are given in section 6. Concluding remarks appear in section 7.

2 The Seuss Programming Notation

The central construct in Seuss is **box**; it plays the role of an object. A program consists of a set of boxes. Typically, a user defines generic boxes, called **cat**s (*cat* is short for *category*), and creates several boxes from each cat through instantiation. A cat is similar to a class; a box is similar to a class instance.

The state of a box is given by the values of its variables. The variables are local to the box. Therefore, their values can be changed only by the steps taken within the box. To enable other boxes to change the state of a box, each box includes a set of **procedures** that may be called from outside. Procedure call is the only mechanism for interaction among boxes.

A procedure is either an **action** or a **method**. A method is called by a procedure of another box. An action is not called like a traditional procedure; it is executed from time to time under the following fairness rule: each action is executed eventually. Both actions and methods can change the state (values of the variables) of their own box, and, possibly, of other boxes by calling their methods. A method may have parameters; an action does not have any parameter.

A method may *accept* or *reject* a call made upon it. If the state of the box does not permit a method to execute —for instance, a *get* method on a channel can not execute if the channel is empty— then the call is rejected. Otherwise, the call is accepted. Some methods accept every call; such methods are called **total** methods. A method that may reject a call is called a **partial** method. Similarly, we have total and partial actions.

2.1 Seuss Syntax

In this section, we introduce a notation for writing programs. 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 (there are different implementations with C++ and Java as host languages) and syntactic aspects that are unrelated to the model are left unspecified in the notation.

Notational Conventions

The notation is described using BNF. All non-terminal identifiers are in Roman and all terminal identifiers are in boldface type. The traditional meta symbols of BNF — ::= { } [ ] ( ) — are used, along with $\vee$ to stand for alternation (the usual symbol for alternation, "|", is a terminal symbol in our notation). The special symbols used as terminals are | // ; : :: in the syntax given below. A syntactic unit enclosed within "{" and "}" in a production may be instantiated zero or more times, and a unit within "[" and "]" may be instantiated zero or one time. 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. Text enclosed within " { " and " } " in a program is to be treated as a comment.

Program

program ::= **program** program-name {cat $\vee$ box} **end**
cat ::= **cat** cat-name [parameters]: {variable} {procedure} **end**
box ::= **box** box-name [parameters]: cat-name

A program consists of a set of cats and boxes in any order. The declaration of a cat or box includes its name and, possibly, parameters. The names of programs, cats and boxes are identifiers. The parameters of a cat or box could be ordinary variables, cats or boxes. A cat consists of (zero or more) variable declarations followed by procedure declarations. A box is an instance of a cat. Variables are declared and initialized in a cat as in traditional programming languages.

Example

We use a single running example to illustrate the syntax of Seuss. A ubiquitous concept in multiprogramming is a *Semaphore*. The skeletal program given below includes a definition of *Semaphore* as a cat and two instances of *Semaphore*, s and t. Cat *user* describes a group of users that execute their critical sections only if they hold both semaphores, s, t; there are three instances of *user*.

---

program *MutualExclusion*
  cat *Semaphore*
    var n: nat **init** 1 {initially, the semaphore value is 1}
    {The procedures of *Semaphore* are to be included here}
  end {*Semaphore*}

  box s, t : *Semaphore*

  cat *user*
    var hs, ht: boolean **init** *false*
    {hs is *true* when *user* holds s. Similarly, ht.}
    {The procedures of *user* are to be included here}
  end {*user*}

  box u, v, w : *user*
end {*MutualExclusion*}

---

procedure

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** [label] :: partial-body
total-method ::= **method** head :: total-body
total-action ::= **action** [label] :: 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 method has a head and a body. The 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 labels are optional for actions; they have no effect on program execution.

Example (contd.)

We add the procedure names to the previous skeletal program.

---

program *MutualExclusion*
  cat *Semaphore*
    var n: nat **init** 1 {initially, the semaphore value is 1}
    partial method P:: { Body of P goes here}
    total method V:: { Body of V goes here}
  end {*Semaphore*}

box s, t : *Semaphore*

cat *user*
var hs, ht: boolean **init** *false*
partial action $s.acquire$:: {acquire s and set hs *true.*}
partial action $t.acquire$:: {acquire s and set hs *true.*}
partial action *execute*::
{Execute this body if both hs, ht are *true.* Then, set hs, ht *false.*}
end {*user*}

box u, v, w : *user*
end {*MutualExclusion*}

---

procedure body

A procedure body has different forms for partial and total procedures. For this manuscript, we take a total-body to be any sequential program. The partial-body is defined by:

partial-body ::= alternative {(| alternative) ∨ (⊬ alternative)}
alternative ::= pre-condition [; pre-procedure] → total-body
pre-condition ::= predicate
pre-procedure ::= partial-method-call

The body of a partial procedure consists of one or more alternatives. Each alternative has a pre-condition, an optional pre-procedure and a total-body. A pre-condition is a predicate on the state of the box to which this procedure belongs (i.e., it is constrained to name only the local variables of the box in which the procedure appears). A pre-procedure is a call upon a partial method (in some other box).

Example (contd.)

Below, we include code for each procedure body. The partial actions $s.acquire$ and $t.acquire$ in *user* include calls upon the partial methods $s.P$ and $t.P$ as pre-procedures. The partial action *execute* in *user* calls the total methods $s.V$ and $t.V$ in its body. The partial action P in *Semaphore* has no pre-procedure.

---

program *MutualExclusion*
cat *Semaphore*
var n: nat **init** 1 {initially, the semaphore value is 1}
partial method P:: $n > 0 \rightarrow n := n - 1$
total method V:: $n := n + 1$

```
  end {Semaphore}

  box s, t : Semaphore

  cat user
    var hs, ht: boolean init false
    partial action s.acquire:: ¬hs; s.P → hs := true
    partial action t.acquire:: ¬ht; t.P → ht := true
    partial action execute::
      hs ∧ ht → critical section; s.V; t.V; hs := false; ht := false
  end {user}

  box u, v, w : user
end {MutualExclusion}
```

The operational semantics of Seuss programs is described in section 2.2. The program, given above, may become deadlocked, that is, it may not allow any user to enter its critical section because one may have acquired s and another t. This problem may be avoided by acquiring s, t in order (i.e., by changing the pre-condition of $t.acquire$ to $hs \wedge \neg ht$).

Multiple Alternatives

Each alternative in a partial procedure is *positive* or *negative*: the first alternative is always positive; an alternative preceded by | is positive and one preceded by ⧸ is negative. For each partial procedure at most one of its alternatives holds in any state; that is, the pre-conditions in the alternatives of a partial procedure are pairwise disjoint. The distinction between positive and negative alternatives is explained under the operational semantics of Seuss in section 2.2.

Restrictions on Programs

Procedure Call

A total-body can include a call only to a total method; a partial method cannot be called by a total body. A partial method can only appear as a pre-procedure in an alternative of a partial procedure. The syntax specifies that an alternative can have at most one pre-procedure. In the example in page 6, partial action $s.acquire$ calls $s.P$ as a pre-procedure, and $execute$ calls the total methods $s.V$, $t.V$ in its total body (i.e., in the code following $\rightarrow$).

Partial Order on Boxes

See section 4.1.

Termination Condition

Execution of each total body (the body part of any action, total or partial) must terminate; this is a proof obligation that has to be discharged by the programmer.

The termination condition can be proven by induction on the "level" of a procedure. First, show that any procedure that calls no other procedure terminates whenever it accepts a call. Next, show that execution of any procedure p terminates assuming that executions of all procedures that p calls terminate.

2.2 Seuss Semantics (Operational)

At run time, a program consists of a set of boxes; their states are initialized at the beginning of the run. There are two different execution styles for a program. In a *tight execution* one action is executed at a time. There is no notion of concurrent execution; each action completes before the next action is started. In a *loose execution* actions may be executed concurrently.

The programmer understands a program by reasoning about its tight executions only. We have developed a logic for this reasoning. An implementation may choose a loose execution for a program in order to maximize resource utilization.

Tight Execution

A tight execution consists of an infinite number of steps; in each step, an action of a box is chosen and executed as described below (in section 2.2). The choice of action to execute in a step is arbitrary except for the following fairness constraint: each action of each box is chosen eventually.

Observe that methods are executed only when they are called from other methods or actions, though actions execute autonomously (and eventually).

Procedure Execution

A method is executed when it is called. To simplify description, we imagine that an action is called by a *scheduler*. Then the distinction between a method and an action vanishes; each procedure is executed when called.

A procedure *accepts* or *rejects* a call. A total procedure always *accepts* calls; its body is executed whenever it is called. Termination condition (see section 2.1) ensures that execution of each total procedure terminates. A partial procedure may *accept* or *reject* a call. Consider a partial procedure g that consists of a single (positive) alternative; then, g is of the following form:

partial method $g(x, y)$:: p; $h(u, v) \rightarrow S$

Execution of g can be described by the following rules.

---

if $\neg p$ then *reject*
else {p holds} call h with parameters (u, v);
  if h *rejects* then *reject*
  else {h *accepts*}
    execute S using parameters, if any, returned by h;
    return parameters, if any, to the caller of g and *accept*
  endif
endif

As stated earlier, the programmer must ensure that execution of each total procedure terminates. It can be then be shown that the execution of any partial procedure g terminates, by using induction on the partial order induced by $\geq_g$ (see section 2.1).

The caller is oblivious to rejection, because then its body is not executed and its state remains unchanged. If all alternatives in a program are positive, then the effect of execution of an action is either rejection (then the state does not change for any box) or acceptance (some box state may change then). This is because, if any procedure rejects during the execution of an action then the entire action rejects. If any procedure accepts —the lowest procedure, that has no pre-procedure, accepts first, followed by acceptances by its callers in the reverse order of calls— then the entire action accepts. This execution strategy meets the *commit* requirement in databases where a transaction either executes to completion or does not execute at all.

We have described the execution of a partial procedure that has a single (positive) alternative. In case a procedure has several alternatives, positive and negative, the following execution strategy is adopted. Recall that pre-conditions of the alternatives are disjoint.

---

if pre-conditions of all alternatives are *false* then *reject*
else {pre-condition of exactly one alternative, f, holds}
  if f is a positive alternative then execute as described previously
  else {f is a negative alternative}
    execute f as a positive alternative except on completion of f:
      reject the call and do not return parameter values
  endif
endif

The execution of a negative alternative always results in rejection. The caller is still oblivious to rejection, because its body is not executed and

its state remains unchanged. However, a called method may change the state of its own box even when it rejects a call, by executing a negative alternative.

For a partial action the effect of execution is identical for positive and negative alternatives because the scheduler does not discriminate between acceptance and rejection of an action. Therefore, partial actions, generally, have no negative alternatives.

3 A Model of Seuss Programs

In this section, we formalize the notion of box, procedure, and executions of procedures (program execution is treated in section 6). The *cats* of Seuss are not modeled because they have no relevance at run time. Also, we do not distinguish between action and method because this distinction is unnecessary for the proof of the theorem. Negative alternatives are not considered in the rest of this paper.

- A *box* is a pair (S, P) where
  S is a set of *states* and
  P is a set of *procedures*.

  Each procedure has a unique name and is designated either *partial* or *total*.

- A *procedure* is a tuple (T, N, E) where
  T is a set of *terminal* symbols; each is a binary relation over the states of its box.
  N is a set of *nonterminal* symbols; each is the name of a procedure of another box.
  E is a non-empty set of *executions*, where each execution is a finite string over $T \cup N$.

  An execution of a total procedure is a sequence where each element of the sequence is either a terminal or a total procedure of another box. An execution of a partial procedure is of the form: $b\ h\ e$, where b is a terminal, h —which is optional— is a nonterminal that names a partial procedure of another box, and e is a sequence in which each element is either a terminal or a total procedure of another box.

- A *program* is a finite set of boxes. Program state is given by the box states. (Therefore, each terminal symbol is a binary relation over the program states.)

Convention and Notation:

(1) Terminal symbols of different procedures are distinct.
(2) Each execution of procedure p begins with a $begin_p$ symbol and ends

with a end_p symbol. Both of these are terminal symbols of procedure p.
(3) For terminal s, $s.box$ is the box of which s is a symbol. Similarly, $p.box$ is defined for a procedure p.

Justification for the Model

A terminal symbol of a procedure —an element of T— denotes a local step within the procedure. The local step can affect only the state of the corresponding box, and we allow a step to have nondeterministic outcome. Hence, each terminal is modeled as a binary relation over box states.

In the formal model, procedures are parameter-less. Although this would be an absurd assumption in practice, it simplifies mathematical modeling considerably. We justify this assumption as follows. First, we can remove a value parameter from a procedure by creating a set of procedures, one for each possible value of the parameter, and the caller can decide which procedure to call based on the parameter value. Thus, all value parameters may be removed at the expense of increasing the set of procedures. Next, consider a procedure with result parameters; to be specific, let $read(w)$ return a boolean value in w. The caller of *read* cannot decide a priori what the returned value will be. However, we can remove parameter w, as follows. First, model *read* by two different procedures, *readt* and *readf*, which return the values *true* and *false*, respectively. Now, we have two different execution fragments modeling the call upon $read(w)$:

readt; $w :=$ *true*, and
readf; $w :=$ *false*.

An execution that calls $read(w)$ will be represented by two executions in our model, one for each possible value returned by *read* for w. Thus, we can remove all parameters from procedures.

Next, we justify our model of procedure execution. An execution is a sequence of steps taken by a procedure and the procedures it calls. To motivate further discussion, consider a procedure P that calls $read(w)$, described above, twice in succession. The terminal symbols of P are α, β where

α denotes $w :=$ *true*, and β denotes $w :=$ *false*.

The nonterminals of P are *readt* and *readf*, as described above.

An execution of P does the following steps twice: call *read* and then assign the value returned in the parameter to w. If P is executed alone then the possible executions are

$begin_P$ *readt* α *readt* α end_P, and
$begin_P$ *readf* β *readf* β end_P.

These are the *tight* executions of P. If, however, other procedures execute concurrently with P then the value of the boolean could change in between the two read operations (by other concurrently executing procedures) and the loose executions of P are:

$begin_P$ *readt* α *readt* α end_P,

$begin_P\ readf\ \beta\ readf\ \beta\ end_P$,
$begin_P\ readt\ \alpha\ readf\ \beta\ end_P$, and
$begin_P\ readf\ \beta\ readt\ \alpha\ end_P$.
In particular, the execution $begin_P\ readt\ \alpha\ readf\ \beta\ end_P$ denotes that the boolean value is changed from *true* to *false* by another procedure during the two calls to *read* by P. Our goal is to model concurrent executions; therefore, we admit all four executions, shown above, as possible executions of P.

We have not specified the initial states of the boxes, because we do not need the initial states to prove the main theorem.

4 Restrictions on Programs

We impose two restrictions on programs.

- (Partial Order on Boxes) For each procedure, there is a partial order over the boxes of the program such that during execution of that procedure, one procedure may call another only if the former belongs to a higher box than the latter; see section 4.1. Different procedures may impose different partial orders on the boxes. A static partial order (i.e., one that is the same for all procedures) is inadequate in practice.

  A consequence of the requirement of partial order is that if some procedure of a box is executing then no procedure of that box is called; therefore, at most one procedure from any box is executing at any moment.

- (Box Condition) For any box, at most one of its procedures may execute at any time; see section 4.3. This restriction disallows concurrency within a box.

4.1 Partial Order on Boxes

Definition:

For procedures p, q, we write $p\ calls\ q$ to mean that p has q as a nonterminal. Let $calls^+$ be the transitive closure of $calls$, and $calls^*$ the reflexive transitive closure of $calls$. Define a relation $calls_p$ over procedures where
$(x\ calls_p\ y) \equiv (p\ calls^*\ x) \wedge (x\ calls\ y)$.

In operational terms, $x\ calls_p\ y$ means procedure x may call procedure y in some execution of procedure p. Each program is required to satisfy the following condition.

Partial Order on Boxes:

For every procedure p, there is a partial order $\geq_p$ over the boxes such that $x\ calls_p\ y \Rightarrow x.box >_p y.box$.

Note: $b >_p c$ is a shorthand for $b \geq_p c\ \wedge\ b \neq c$. Relation $\geq_p$ is reflexive and $>_p$ is irreflexive.

Observation 1:

$p\ calls^*\ x \Rightarrow p.box \geq_p x.box$, and
$p\ calls^+\ x \Rightarrow p.box >_p x.box$.

Proof: Define $calls^i$, for $i \geq 0$, as follows.

$p\ calls^0\ p$, and
$p\ calls^{i+1}\ q \equiv (\exists r :: p\ calls^i\ r\ \wedge\ r\ calls\ q)$.

Using induction over i we can show that

$p\ calls^i\ x \Rightarrow p.box \geq_p x.box$, for all i, $i \geq 0$
$p\ calls^i\ x \Rightarrow p.box >_p x.box$, for all i, $i > 0$.

The desired results follow by noting that

$p\ calls^*\ x \equiv (\exists i : i \geq 0 : p\ calls^i\ x)$, and
$p\ calls^+\ x \equiv (\exists i : i > 0 : p\ calls^i\ x)$. □

Note that $p\ calls^+\ q\ \Rightarrow$ {by Observation 1} $(p.box >_p q.box)\ \Rightarrow\ p, q$ are in different boxes. It follows that no call is ever made upon a box when one of its procedures has started but not completed its execution.

Observation 2:

$calls^+$ is an acyclic (i.e., irreflexive, asymmetric and transitive) relation over the procedures.
Proof: From its definition $calls^+$ is transitive. Also, $p\ calls^+\ p\ \Rightarrow$ {from Observation 1} $p.box >_p p.box$, a contradiction. Therefore, $calls^+$ is irreflexive. Asymmetry of $calls^+$ follows similarly.

Definition:

The *height* of a procedure is a natural number. The height is 0 if the procedure has no nonterminal. Otherwise, $p\ calls\ q \Rightarrow p.height > q.height$. This definition of height is well grounded because $calls^+$ induces an acyclic relation on the procedures.

Definition:

An *execution tree* of procedure p is an ordered tree where (1) the root is labeled p, (2) every nonleaf node is labeled with a nonterminal symbol, and (3) the sequence of labels of the children of a nonleaf node q is an execution

of q. A *full execution tree* is an execution tree in which each leaf node is labeled with a terminal symbol.

Any execution tree of procedure p is finite. This is because if procedure q is an ancestor of procedure r in this tree then $q\ calls_p\ r$; hence, $q.box >_p r.box$. Since the program has a finite number of boxes, each path in the tree is finite; also, the degree of each node is finite because each execution is finite in length. From Koenig's lemma, the tree is finite.

Definition:

The *frontier* of an execution tree is the ordered sequence of symbols in the leaf nodes of the tree. An *expanded execution* of procedure p is the frontier of some full execution tree of p. Hence, an expanded execution consists of terminals only.

4.2 Procedures as Relations

With each terminal symbol we have associated a binary relation over program states. Next, we associate such a relation with each procedure and each execution of a procedure; to simplify notation we use the same symbol for an execution (or a procedure) and its associated relation. For execution e, $(u, v) \in e$ means that if e is started in state u then it is possible for it to end in state v. For a procedure p, $(u, v) \in p$ means that there is an execution e of p such that $(u, v) \in e$. Formally,

- The relation for a procedure is the union of relations of all its executions.
- The relation for an execution $x_0, \ldots, x_n$ is the relational product of the sequence of relations corresponding to the x_i's.

Observe that a symbol x_i in an execution may be a terminal for which the relation has already been defined, or a nonterminal for which the relation has to be computed using this definition. We show in the following lemma that the rules given above define unique relations for each execution and procedure; the key to the proof is the acyclicity of $calls^+$.

Lemma 1:

There is a unique relation for each procedure and each execution.

Proof: We prove the result by induction on n, the height of a procedure.

For $n = 0$: The procedure has only terminals in all its executions. The relation associated with any execution of the procedure is the relational product of its terminals. The relation associated with the procedure is the union of all its executions, and, hence, is uniquely determined.

For $n > 0$: Each execution of the procedure has terminals (for which the relations are given) or nonterminals (whose heights are at most n, and,

hence, they have unique relations associated with them). Therefore, the relation for an execution —which is the relational product of the sequence of relations of its terminals and nonterminals— is uniquely determined. So, the relation for the procedure is also uniquely determined. □

Note that an execution may have the empty relation associated with it, denoting that the steps of the execution will never appear contiguously in a program execution. Such is the case with the execution $read\ \alpha\ read\ \beta$ in the example of section 3, where two successive reads of the same variable yield different values. Such an execution may appear as a noncontiguous subsequence in a program execution where steps of another procedure's execution could alter the value of the variable in between the two read operations.

Henceforth, each symbol —terminal or nonterminal— has an associated binary relation over program states. Concatenation of symbols corresponds to their relational product. For strings x, y, we write $x \subseteq y$ to denote that the relation corresponding to x is a subset of the relation corresponding to y.

Observation 3:

For terminal symbols s, t of different boxes, $st = ts$ (i.e., the relations st and ts are identical).

4.3 Box Condition

The execution strategy for a program ensures that at most one procedure from a box executes at any time. This strategy can be encoded in our model by making it impossible for procedure q to start if procedure p of the same box has started and not yet completed. This is formalized below.

Definition:

Let σ and τ be sequences of symbols (terminals and nonterminals). Procedure p is *incomplete* after σ (before τ in $\sigma\tau$) if σ contains fewer end_p's than $begin_p$'s.

Box Condition

Let p, q be procedures of the same box, and p be incomplete after σ. Then, $\sigma\ begin_q = \epsilon$, where ϵ denotes the empty relation.

The following lemma shows that under certain conditions a terminal symbol can be transposed with a nonterminal symbol adjacent to it .

Lemma 2:

Let p, q be procedures, t a terminal of p, and σ any sequence of symbols.
1. If p is incomplete after σ then $\sigma\ q\ t\ \subseteq\ \sigma\ t\ q$.

2. If p is incomplete after $\sigma\ t$ then $\sigma\ t\ q\ \subseteq\ \sigma\ q\ t$.
Proof: We prove the first part. The other part is left to the reader.

$$
\begin{array}{ll}
 & \sigma\ q\ t \\
= & \{q \text{ is the union of all its expanded executions, } g\} \\
 & (\cup_g(\sigma\ g\ t)) \\
= & \{\text{partition } g \text{ into } e, f;\ e \text{ has a terminal from } p.box, \text{ and } f \text{ does not}\} \\
 & (\cup_e(\sigma\ e\ t)) \cup (\cup_f(\sigma\ f\ t)) \\
= & \{e \text{ is of the form } \sigma' begin_r\ \sigma'', \text{ where:} \\
 & \quad \sigma' \text{ has no terminal from } p.box;\ r \text{ is some procedure from } p.box\} \\
 & (\cup(\sigma\sigma'\ begin_r\ \sigma''\ t)) \cup (\cup_f(\sigma\ f\ t)) \\
= & \{\sigma\sigma'\ begin_r = \epsilon, \text{ because from Box Condition:} \\
 & \quad p \text{ is incomplete after } \sigma, \text{ and hence, after } \sigma\sigma', \text{ and } r.box = p.box\} \\
 & (\cup_f(\sigma\ f\ t)) \\
= & \{f \text{ has no terminal from } p.box,\ t \text{ is a terminal of } p.box;\ \text{Observation 3}\} \\
 & (\cup_f(\sigma\ t\ f)) \\
\subseteq & \{f \text{ is a subset of the (expanded) executions of } q\} \\
 & \sigma\ t\ q
\end{array}
$$

5 Compatibility

A loose execution of a program allows only compatible actions to be executed concurrently. We give a definition of compatibility in this section. We expect the user to specify the compatibility relation for procedures within each box; then the compatibility relation among all procedures (in different boxes) can be computed automatically in linear time from the definition given below.

Procedures p, q are *compatible*, denoted by $p\ \sim\ q$, if all of the following conditions hold. Observe that $\sim$ is a symmetric relation.

C0. $p\ calls\ p'\ \Rightarrow\ p' \sim q$, and $q\ calls\ q'\ \Rightarrow\ p \sim q'$.

C1. If p, q are in the same box,
(p is total $\Rightarrow\ qp \subseteq pq$), and
(q is total $\Rightarrow\ pq \subseteq qp$).

C2. If p, q are in different boxes, the transitive closure of the relation $(\geq_p \cup \geq_q)$ is a partial order over the boxes.

Condition C0 requires that procedures that are called by compatible procedures be compatible. Condition C1 says that for p, q in the same box, the effect of executing a partial procedure and then a total procedure can be simulated by executing them in the reverse order. Condition C2 says that compatible procedures impose similar (i.e., nonconflicting) partial orders on boxes.

Notes:

(1) If partial procedures p, q of the same box call no other procedure, then they are compatible.
(2) Total procedures p, q of the same box are compatible only if $pq = qp$.
(3) The condition (C0) is well grounded because if p *calls* p' then the height of p exceeds that of p'.
(4) In a Seuss program compatibility of procedures with parameters has to be established by checking the compatibility with all possible values of parameters; see the example of channels in section 5.1

5.1 Examples of Compatibility

Semaphore

Consider the *Semaphore* box of page 6. We show that $V \sim V$ and $P \sim V$, that is,

$$VV = VV \text{ , and}$$
$$PV \subseteq VP$$

The first identity is trivial. For the second identity, we compute the relations corresponding to P and V, as follows:

$$\begin{array}{ll} & P \\ = & \{\text{from the program text}\} \\ & (n > 0) \times (n := n - 1) \\ = & \{\text{definitions of predicate and assignment}\} \\ & \{(x, x)|\ x > 0\} \times \{(x, x - 1)|\ x > 0\} \\ = & \{\text{simplifying}\} \\ & \{(x, x - 1)|\ x > 0\} \end{array}$$

Similarly, $V = \{(x, x + 1)|\ x \geq 0\}$. Taking relational product, $PV = \{(x, x)|\ x > 0\}$, and $VP = \{(x, x)|\ x \geq 0\}$. Therefore, $PV \subseteq VP$.

Channels

Consider the unbounded FIFO channel of sectionthat $get \sim put$, that is, for any x, y,

$$get(x)\ put(y) \subseteq put(y)\ get(x)$$

That is, any state reachable by executing $get(x)\ put(y)$ is also reachable by executing $put(y)\ get(x)$ starting from the same initial state.

Let $(u, v) \in get(x)\ put(y)$. We show that $(u, v) \in put(y)\ get(x)$. Given $(u, v) \in get(x)\ put(y)$, we conclude from the definition of relational composition, that there is a state w such that $(u, w) \in get(x)$ and $(w, v) \in put(y)$. Since $(u, w) \in get(x)$, from the implementation of get, u represents a state

where the channel is non-empty; that is, the channel state s is of the form $a +\!\!+ S$, for some item a and a sequence of items S. Then we have

$$\{s = a +\!\!+ S\}\ put(y)\ \{s = a +\!\!+ S +\!\!+ y\}\ get(x)\ \{x +\!\!+ s = a +\!\!+ S +\!\!+ y\}$$
$$\{s = a +\!\!+ S\}\ get(x)\ \{x +\!\!+ s = a +\!\!+ S\}\ put(y)\ \{x +\!\!+ s = a +\!\!+ S +\!\!+ y\}$$

The final states, given by the values of x and s, are identical. This completes the proof.

The preceding argument shows that two procedures from different boxes that call *put* and *get* (i.e., a sender and a receiver) may execute concurrently. Further, since $get \sim get$ by definition, multiple receivers may also execute concurrently. However, it is not the case that $put \sim put$ for arbitrary x, y, that is,

$$put(x)\ put(y) \neq put(y)\ put(x)$$

because a FIFO channel is a sequence, and appending a pair of items in different orders results in different sequences. Therefore, multiple senders may not execute concurrently.

Next, consider concurrent executions of multiple senders and receivers, as is the case in a client–server type interaction. As we have noted in the last paragraph, multiple senders may not execute concurrently on a FIFO channel. Therefore, we use an unordered channel, of section for communication in this case. We show that $put \sim put$ and $put \sim get$ for unordered channel, i.e., for all x, y

$$put(x)\ put(y)\ =\ put(y)\ put(x)\ \text{, and}$$
$$get(x)\ put(y)\ \subseteq\ put(y)\ get(x)$$

The proof of the first identity is trivial because *put* is implemented as a bag union. The proof of the second result is similar to that for the FIFO channel. We need consider the initial states where the bag b is non-empty. In the following, $x \cup b$ is an abbreviation for $\{x\} \cup b$.

$$\{b = B, B \neq empty\}\ get(x)\ \{x \cup b = B\}\ put(y)\ \{x \in B,\ x \cup b = B \cup y\}$$
$$\{b = B, B \neq empty\}\ put(y)\ \{b = B \cup y\}\ get(x)\ \{x \in (B \cup y),\ x \cup b = B \cup y\}$$

The post-condition of (1) implies the post-condition of (2) because $x \in B \Rightarrow x \in (B \cup y)$. Hence, any final state of $get(x)\ put(y)$ is also a final state of $put(y)\ get(x)$.

5.2 Semi-Commutativity of Compatible Procedures

In Lemma 3, below, we prove a result for compatible procedures analogous to condition C1 of page 16. This result applies to any pair of compatible procedures, not necessarily those in the same box.

Lemma 3:

Let $p \sim q$ where p is total (p, q need not belong to the same box). Then $qp \subseteq pq$.

Proof: We apply induction on n, the sum of the heights of p and q, to prove the result. The result holds from the definition of $\sim$ if p, q are in the same box. Assume, therefore, that p, q are in different boxes.

For $n = 0$: Both p, q are at height 0; hence, p, q have only terminals in all their executions. Since, p, q are from different boxes, the result follows by repeated application of Observation 3.

For $n > 0$: From (C2), the transitive closure of $(\geq_p \cup \geq_q)$ is a partial order over the boxes; we abbreviate this relation by $\geq$. We prove the result for the case where $\neg(q.box > p.box)$. A similar argument applies for the remaining case, $\neg(p.box > q.box)$. Consider an execution, e, of p. Let x be any symbol in that execution. We show that $qx \subseteq xq$.

- x is a terminal: Consider any expanded execution of q. A terminal t in this expanded execution is a symbol of procedure r where $q\ calls^*\ r$.

  $x.box = t.box$
  $\Rightarrow$ {x, t are terminals of p, r, respectively}
  $x.box = t.box\ \wedge\ x.box = p.box\ \wedge\ t.box = r.box$
  $\Rightarrow$ {logic}
  $p.box = r.box$
  $\Rightarrow$ {$q\ calls^*\ r$; Observation 1}
  $p.box = r.box\ \wedge\ q.box \geq_q r.box$
  $\Rightarrow$ {logic}
  $q.box \geq_q p.box$
  $\Rightarrow$ {$\geq$ is the transitive closure of $(\geq_p \cup \geq_q)$}
  $q.box \geq p.box$
  $\Rightarrow$ {p, q are from different boxes}
  $q.box > p.box$
  $\Rightarrow$ {assumption: $\neg(q.box > p.box)$}
  false

  Thus, x, t belong to different boxes, and from Observation 3, $xt = tx$. Applying this argument for all terminals t in the expanded execution of q, we have $qx = xq$.

- x is a nonterminal: From (C0), $x \sim q$. The combined heights of x and q is less than n. Also, x is total, since it is a nonterminal of p, and p is total. From the induction hypothesis, $qx \subseteq xq$.

Next we show that for any execution e of p, $qe \subseteq eq$. Proof is by induction on the length of e. If the length of e is 1 then the result follows from $qx \subseteq xq$. For e of the form fx:

$$
\begin{array}{ll}
 & qfx \\
\subseteq & \{\text{Induction: } qf \subseteq fq\text{; monotonicity of relational product}\} \\
 & fqx \\
\subseteq & \{qx \subseteq xq\text{; monotonicity of relational product}\} \\
 & fxq
\end{array}
$$

Next, we show $qp \subseteq pq$.

$$
\begin{array}{ll}
 & qp \\
= & \{\text{definition of } p\} \\
 & q(\cup_{e \in p}\ e) \\
= & \{\text{distributivity of relational product over union}\} \\
 & (\cup_{e \in p}\ qe) \\
\subseteq & \{qe \subseteq eq \text{ from the above proof}\} \\
 & (\cup_{e \in p}\ eq) \\
= & \{\text{distributivity of relational product over union}\} \\
 & (\cup_{e \in p}\ e)q \\
= & \{\text{definition of } p\} \\
 & pq
\end{array}
$$

□

Lemma 4:

$(p \sim q \wedge p\ calls^*\ p' \wedge\ q\ calls^*\ q') \Rightarrow (p' \sim q')$.

Proof: The result follows from

$$(p \sim q\ \wedge p\ calls^i\ p'\ \wedge\ q\ calls^j\ q') \Rightarrow (p' \sim q')$$

which is proved by induction on $i + j$, $i, j \geq 0$.

6 Proof of the Reduction Theorem

A finite *loose execution* of a program is a finite sequence of steps taken by some of the procedures of the program. The executions of the procedures could be interleaved. A loose execution satisfies: (1) the steps taken by each procedure is an expanded execution of that procedure, and (2) executions of two procedures are interleaved only if they are both part of the execution of the same procedure, or if they are compatible.

In this section, we formally define loose execution of a program and show a scheme to convert a loose execution to a tight execution. The reduction scheme establishes the following theorem.

Reduction Theorem:

Let E be a finite loose execution of a program. There exists a tight execution F of the program such that $E \subseteq F$.

6.1 Loose execution

A loose execution is given by: (1) a finite set of full execution trees (of some of the procedures), and (2) a finite sequence of terminals called a *run*. the relation corresponding to a loose execution is the relational product of the terminals in the run. Each execution tree (henceforth called a tree) depicts the steps of one action in this loose execution, and the run specifies the interleaving of the executed steps. The trees and the run satisfy the conditions M0 and M1, given below.

Condition M0 states that each symbol of the run can be uniquely identified with a leaf node of some tree, and conversely, and that the loose execution contains the procedure executions (the frontiers of the corresponding trees) as subsequences. Since each symbol of the run belongs to a tree we write $x.root$ for the root of the tree that symbol x belongs to.

Condition M1 states that if two procedures are incomplete at any point in the run then they either belong to the same tree (i.e., they are part of the same execution) or they are compatible.

- (**M0**) There is a one-to-one correspondence between the symbols in the run and the leaf nodes of the trees. The subsequence of the run corresponding to symbols from a tree T is the frontier of T.
- (**M1**) Suppose procedure p is incomplete before symbol s in the run. Then, either $p.root = s.root$ or $p.root \sim s.root$.

6.2 Reduction Scheme

Suppose R is the run of some loose execution. We transform run R and the execution trees in stages; let R' denote the transformed run. The transformed run may consist of terminals as well as nonterminals, and its execution trees need not be full (i.e., leaf nodes may have nonterminal labels). We show how to transform the execution trees and the run so that the following invariants are maintained. Note the similarity of N0, N1 with M0, M1.

- (**N0**) There is a one-to-one correspondence between the symbols in the run and the leaf nodes of the trees. The subsequence of the run corresponding to symbols from a tree T is the frontier of T.
- (**N1**) Suppose procedure p is incomplete before symbol s in the run. Then, either $p.root = s.root$ or $p.root \sim s.root$.
- (**N2**) $R \subseteq R'$.

The conditions (N0, N1, N2) are initially satisfied by the given run and the execution trees: N0, N1 follow respectively from M0, M1, and N2 holds because $R = R'$.

The reduction process terminates when there are no *end* symbols in the run; then all symbols are the roots of the trees. This run corresponds to a tight execution, and according to N2, it establishes the reduction theorem. The resulting tight execution can simulate the original loose execution: if the original execution starting in a state u can lead to a final state v then so does the final tight execution.

For a run that contains an *end* symbol, we apply either a *replacement* step or a *transposition* step. Let the first *end* symbol appearing in the run belong to procedure q.

Replacement Step:

If a contiguous subsequence of the run corresponds to the frontier of a subtree rooted at q (then the subsequence is an execution of q) replace the subsequence by the symbol q, and delete the subtree rooted at q (retaining q as a leaf node).

This step preserves N0. N1 also holds because for any symbol x in the execution that is replaced by q, $p.root \sim x.root$ prior to replacement, and $x.root = q.root$. Hence, $p.root \sim q.root$ after the replacement. The relation for a procedure is weaker than for any of its executions; therefore, the replacement step preserves N2.

Transposition step:

If a run has an *end* symbol, and a replacement step is not applicable then execution of some procedure q is noncontiguous. We then apply a transposition step to transpose two adjacent symbols in the run (leaving the execution trees unchanged) that makes the symbols of q more contiguous. Continued transpositions make it possible to apply a replacement step eventually.

Suppose q is a partial procedure (similar arguments apply to partial procedures that have no pre-procedures and to total procedures). An execution of procedure q is of the form $(begin_q\ b\ h\ \ldots\ x\ \ldots\ end_q)$ where h is the pre-procedure of q and x is either a terminal symbol or a nonterminal, designating a total procedure, of q. All procedures that complete before q have already been replaced by nonterminals, because the first *end* symbol appearing in the run belongs to q. Note that h is a procedure that completes before q.

Suppose x is preceded by y which is not part of the execution of q. We show how to bring x closer to h. Transposing x, y preserves N0, N1. We show below that transposition preserves N2, as well.

- Case 0 (Both x, y are terminals): Let y be a terminal of procedures p. Procedure q is incomplete before y because its end_q symbol comes later. If p, q are in the same box then the relation corresponding to prefix σ of the run up to y is ϵ, from the Box condition. Hence,

$\sigma\ y\ x = \sigma\ x\ y$. If p, q belong to different boxes, from Observation 3, the symbols x, y can be transposed.

- Case 1 (Both x, y are nonterminals): Symbol x is part of q's execution; therefore, $q.root\ calls^*\ x$. Symbol y is not a part of q's execution, nor can it be a part of the execution of any procedure that calls q because q is incomplete before y; therefore, $q.root \neq y.root$.

$$
\begin{array}{ll}
 & q \text{ is incomplete just before } y \\
\Rightarrow & \{\text{(N1)}\} \\
 & q.root = y.root \ \vee\ q.root \sim y.root \\
\Rightarrow & \{q.root \neq y.root \text{ (see above)}\} \\
 & q.root \sim y.root \\
\Rightarrow & \{q.root\ calls^*\ x \ \wedge\ y.root\ calls^*\ y\text{; Lemma 4}\} \\
 & x \sim y \\
\Rightarrow & \{x \text{ is total; Lemma 3}\} \\
 & yx \subseteq xy
\end{array}
$$

- Case 2 (x is a terminal, y a nonterminal): q is incomplete just before y. Applying Lemma 2 (part 1), x, y may be transposed.

- Case 3 (x is a nonterminal, y is a terminal): Let Y be the procedure of which y is a symbol. Since the first *end* symbol in the run belongs to q, end_Y comes after x. Therefore, Y is incomplete before x. Applying Lemma 2 (part 2) with Y as the incomplete procedure, x, y may be transposed.

Thus, x, y may be transposed in all cases, preserving N3. Hence, all symbols in the execution of q to the right of h can be brought next to h.

Next, we bring the $begin_q$ symbol and the predicate b next to h, using an argument similar to Case 3, above. Thus, all of q's symbols to the left and right of h can be made contiguous around h, and a replacement step can then be applied.

For a total procedure q the reduction is done similarly; $begin_q$ serves the role of h in the above argument. For a procedure q that has no pre-procedure, the reduction process is similar with b serving the role of h.

Proof of Termination of the Reduction Scheme

We show that only a finite number of replacement and transposition steps can be applied to any loose execution. For a given run, consider the procedure q whose *end* symbol, end_q, is the first *end* symbol in the run. Define two parameters of the run, n, c, as follows.

$n =$ number of end symbols in the run,
$c = \Sigma\ c_j$,

where c_j is the number of symbols not belonging to q between the preprocedure h of q and the j^{th} symbol of q, and the sum is over all symbols of q. c has an arbitrary value if the run has no *end* symbol.

The pair (n, c) decreases lexicographically with each transposition and replacement step. This is because a replacement step removes one end symbol from the run, thus decreasing n. A transposition step decreases c while keeping n unchanged. Ultimately, therefore, n will become 0; then the run has no *end* symbol, and, from (N0), the symbols are the roots of the execution trees.

7 Concluding Remarks

The following variation of the Reduction theorem may be useful for applications on the worldwide web. Consider a Seuss program in which every procedure calls at most one other procedure. Define all pairs of procedures to be compatible. The reduction theorem then holds: any loose execution may be simulated by some tight execution.

The proof of this result is similar to the proof already given. As before, we reduce procedure q, whose end symbol, end_q, is the first end symbol in the run. If this procedure calls no other procedure, then all its symbols are terminals, and, by applying Case (0) and Case (2) of the transposition step, we can bring all its symbols together next to its first symbol. If the procedure calls another procedure then, according to the reduction procedure, the called procedure has already been reduced and we bring all the symbols next to the called procedure symbol in a similar fashion.

The major simplification in the reduction scheme for this special case is due to the fact that it is never necessary to transpose two nonterminals. Therefore, Case (1) of the transposition step never arises. Consequently, the condition for compatibility of two procedures (page 16) is irrelevant in this case.

Acknowledgments

This paper owes a great deal to discussions with Rajeev Joshi and Will Adams. I am grateful to Carroll Morgan who gave me useful comments on an earlier draft. Ernie Cohen has taught me a great deal about reduction theorems, in general.

8 References

[1] Ernie Cohen and Leslie Lamport. Reduction in TLA. In David Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998. Compaq SRC Research Note 1998-005.

[2] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report 44, Digital Systems Research Center, May 1989.

[3] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[4] Jayadev Misra. Loosely coupled processes. *Future Generation Computer Systems*, 8:269–286, 1992. North-Holland.

[5] Jayadev Misra. *A Discipline of Multiprogramming.* Monographs in Computer Science. Springer-Verlag New York Inc., New York, 2001. The first chapter is available at `http://www.cs.utexas.edu/users/psp/discipline.ps.gz`.