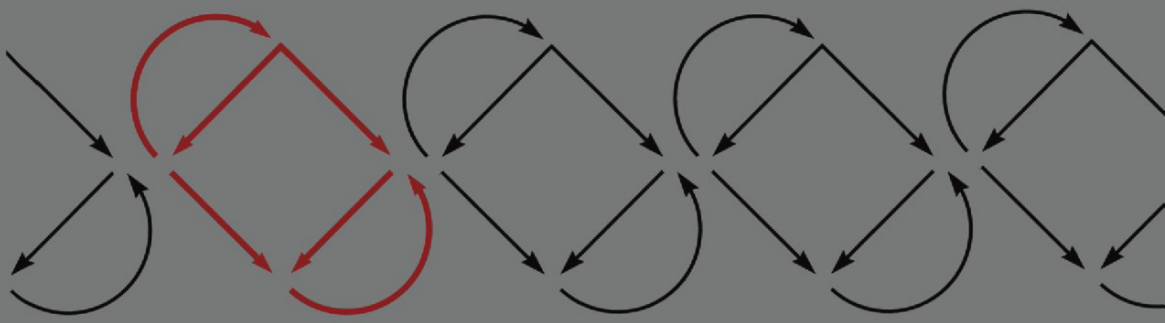
MONOGRAPHS IN COMPUTER SCIENCE

A DISCIPLINE OF MULTIPROGRAMMING

Programming Theory for Distributed Applications

Jayadev Misra



Springer

Jayadev Misra

A Discipline of Multiprogramming

Programming Theory for Distributed Applications

Springer

Jayadev Misra
Department of Computer Sciences
University of Texas
Austin, TX 78712-1188
USA

Library of Congress Cataloging-in-Publication Data
Misra, Jayadev.
A discipline of multiprogramming: programming theory for distributed applications / Jayadev Misra.
p.cm.
Includes bibliographical references and index.

ISBN 978-1-4612-6427-9 ISBN 978-1-4419-8528-6 (eBook)
DOI 10.1007/978-1-4419-8528-6
1. Multiprogramming (Electronic computers) I. Title.
QA76.6 .M528 2001
005.4'34—dc21 2001018392

Printed on acid-free paper.

Production managed by A. Orrantia; manufacturing supervised by Jeffrey Taub.
Photocomposed copy prepared from the author's LaTeX files.

9 8 7 6 5 4 3 2 1

ISBN 978-1-4612-6427-9 SPIN 10790966

Contents

1

A Discipline of Multiprogramming

1.1 Wide-Area Computing

The main software challenge in developing application programs during the 1960s and the 1970s was that the programs had to operate within limited resources, i.e., slow processors, small memories, and limited disk capacities. Application programming became far more widespread during the 1980s because of the falling prices of hardware (which meant that more processing power and storage were available for the same cost) and a better understanding of the application programming process. However, most applications still ran on mainframes or over a cluster of machines in a local-area network; truly distributed applications that ran over wide-area networks were few because of the latency and bandwidth limitations of long-haul communication. The 1990s saw great strides in broad-band communication, and the World Wide Web provides a giant repository of information. This combination promises development of a new generation of distributed applications, ranging from mundane office tasks —e.g., planning a meeting by reading the calendars of the participants— to real-time distributed control and coordination of hundreds of machines —e.g., as would be required in a recovery effort from an earthquake.[1]

The obvious problems in applications design that are related to the characteristics of wide-area communication are security and fault-tolerance.

[1] I am indebted to my colleague Harrick Vin for this example and extensive discussions on related topics.

These issues were present even when most computing was done on a single processor, but they have been magnified because messages can be intercepted more easily over a wide-area network, and it is more likely that some node will fail in a 1,000-node network. We contend that growth in applications programming is hindered *only slightly* by these technical problems; the crucial barrier is that *distributed application design is an extremely difficult task because it embodies many of the complexities associated with concurrent programming.*

The distributed applications we envisage have the structure that they collect data from a number of sources, compute for a while, and then distribute the results to certain destinations. This simple paradigm hides a multitude of issues. When should an application start executing— when invoked by a human, by another application, periodically, say, at midnight, or triggered by an event, say, upon detection of the failure of a communication link? How does an application ensure that the data it accesses during a computation is not altered by another concurrently executing application? How do communicating parties agree on the structure of the data being communicated? How are conflicts in a concurrent computation arbitrated? In short, the basic issues of concurrent computing, such as exclusive access to resources, deadlock, and starvation, and maintaining consistent copies of data, have to be revisited in the wide-area context.

One set of issues arises from the current structure of the World Wide Web. The Web sites are designed today under the assumption that their users are humans, not machines. Therefore, the sites are suitable for navigation by humans, and the browsers make it pleasant —by permitting clicks on hyper-links, for instance— for humans to visit related sites from a given site. The emphasis on human interaction has made it difficult, unfortunately, for machines to extract data from one or more sites, compute, and distribute the results to a number of users. For instance, given a database of news sites, it is not easy to "display all stories about cyclones published in the last 3 days". Given that professors in a department produce a grade sheet for each course they teach, it is currently a major effort to collate this information and produce the grade sheets for all students. Nor is it easy to arrange a meeting of professors all of whose calendars are available online.

Proposal for a programming model

There seems to be an obvious methodology for designing distributed applications: represent each device (computer, robot, a site in the World Wide Web) by an object and have the objects communicate by messages or by calling each others' methods. This representation maps conveniently to the underlying hardware, and it induces a natural partition on the problem that is amenable to stepwise refinement. We start with this model as the basis, and simplify and enhance it so that it is possible to address the concurrent programming issues.

The current view of wide-area programming typically requires a human being to invoke a method, and the method provides its results in a form suitable for human consumption. A program that runs each week to plan a meeting of a set of professors —by scanning their calendars, reserving a room for that time, and notifying the affected parties— is quite cumbersome to design today (see the example in section 1.2). Such programs that run autonomously based on certain conditions —once a week, whenever a grade is posted for a student, or when the stock market crashes— are called *actions* in this book. The coding of methods and actions are essentially identical, and we treat them similarly in the programming model.

We espouse a more elaborate view of methods (and actions) that is appropriate for wide-area computing. It may not always be possible for a method to be executed because the state of the object may not permit it. Such is the case for a *P*-method on a semaphore [58] when the semaphore value is zero, or a monitor [90] procedure that is called to remove an item from a buffer when the buffer is empty. The traditional approach then is to queue the caller, accept calls on other methods that may change the object state, and complete a queued call only when the object state permits it. Therefore, it is possible for a caller to be queued indefinitely.

We adopt a different approach: a call should be *accepted* by a method only if its completion is guaranteed, and *rejected* otherwise; a rejected caller may attempt its call in the future. Callers are not queued, and each caller is guaranteed a response from the called procedure in finite time.

The programming model proposed in this book and the associated theory have been christened *Seuss*. The major goal of Seuss is to simplify multiprogramming[2]. To this end, we separate the concern of concurrent implementation from the core program design problem. A program execution is understood as a single thread of control —sequential executions of actions that are chosen according to some scheduling policy— yet program implementation permits concurrent executions of multiple threads (i.e., actions). As a consequence, it is possible to reason about the properties of a program from its single execution thread, whereas an implementation may exploit the inherent concurrency for efficient execution. A central theorem establishes that multiple execution threads implement single execution threads; i.e., for any concurrent execution of actions there exists an equivalent serial execution of those actions.

The programming model is minimal; all well-known constructs of concurrent programming —process, message communication, synchronization, rendezvous, waiting, sharing, and mutual exclusion— are absent. However, the built-in primitives are powerful enough to encode all known communication and synchronization protocols succinctly. The fundamental concepts

[2]We use the terms "multiprogramming" and "concurrent programming" synonymously.

in the model are objects and procedures; a procedure is a method or an action. No specific communication or synchronization mechanism, except procedure call, is built in.

Seuss proposes a complete disentanglement of the sequential and concurrent aspects of programming. We expect large sections of concurrent programs to be designed, understood, and reasoned about as sequential programs. *A concurrent program merely orchestrates executions of its constituent sequential programs, by specifying the conditions under which each sequential program is to be executed.*

1.2 An Example: Planning a Meeting

To illustrate the intricacies of concurrent programming and motivate discussion of the programming model, we consider a small though realistic example.

1.2.1 Problem description

Professors in a university have to plan meetings from time to time. Each meeting involves a nonempty set P of professors; the meeting has to be held in one of a specified set R of rooms. A meeting can be held at time t provided that *all* members of P can meet at t and *some* room in R is free at t. Henceforth, time is a natural number and each meeting lasts one unit of time. The calendar of professor p can be retrieved by calling procedure $p.next$ with a time value as argument: $p.next(t)$ is the earliest time at or after t when p can meet. Similarly, for room r, $r.next(t)$ is the earliest time at or after t when r is free. Thus, $p.next(t) = t$ denotes that p can meet at t, and there is a similar interpretation of $r.next(t) = t$.

Our goal is to write a procedure *plan* that returns the earliest meeting time within an interval $[L, U)$, where the interval includes L and excludes U, given P and R as arguments; if no such meeting time exists, that fact is reported. Once a suitable meeting time and the associated room are determined, the calendars of the affected professors and the room are changed to reflect that they are busy at that time. To this end, each professor or room x has a procedure $x.reserve$; calling $x.reserve(t)$ reserves x for a meeting at t.

A simpler version of this problem appears in [32, section 1.4] and is also treated in sections 5.5.2 and 6.5.2 of this book. In these versions, room allocation is not a constraint. In the current version, professors impose a universal constraint —*all* professors in P have to meet at the scheduled time— and the rooms impose an existential constraint —*some* room in R should be free then.

1.2.2 *Program development*

Assume that rooms are represented by integers so that they can be numerically compared. Also, for any professor or room x, $x.next$ is ascending and monotonic; i.e., for all times s and t,

$$t \leq x.next(t), \text{ and}$$
$$s \leq t \;\Rightarrow\; x.next(s) \leq x.next(t)\ .$$

See section 5.5.2 for a discussion of these requirements.

Notation The notations for arithmetic and boolean expressions used in this example are explained in appendix A.2.1. In this section

$\langle \forall x :\ x \in P :\ t = x.next(t) \rangle$
  means that all professors in P can meet at t,
$\langle \exists y :\ y \in R :\ t = y.next(t) \rangle$
  means that some room in R is free at t,
$\langle max\ p :\ p \in P :\ p.next(t) \rangle$
  is the maximum over all p in P of $p.next(t)$, and
$\langle min\ y :\ y \in R \wedge t = y.next(t) :\ y \rangle$
  is the smallest (numbered) room in R that is free at t.
  The value of the expression is ∞ if no room is free at t. □

Define time t to be a *common meeting time* (abbreviated to *com*) if all professors in P can meet and some room in R is free at t. That is,

$$com(t) \;\equiv\; \langle \forall x :\ x \in P :\ t = x.next(t) \rangle \;\wedge\; \langle \exists y :\ y \in R :\ t = y.next(t) \rangle.$$

Note that,

$$\langle \forall x :\ x \in P :\ t = x.next(t) \rangle \;\equiv\; (t = \langle max\ p :\ p \in P :\ p.next(t) \rangle).$$

Similarly,

$$\langle \exists y :\ y \in R :\ t = y.next(t) \rangle \equiv\; (t = \langle min\ r :\ r \in R :\ r.next(t) \rangle).$$

Therefore,

$$com(t) \;\equiv\; t = \langle max\ p :\ p \in P :\ p.next(t) \rangle \;\wedge\; t = \langle min\ r :\ r \in R :\ r.next(t) \rangle.$$

In the following procedure, variable t is repeatedly assigned values of the expressions in the two given conjuncts of $com(t)$ (in a specific order, though any order would do) until $com(t)$ holds or t falls outside the interval $[L, U)$. If there is a common meeting time in $[L, U)$, then t is set to the earliest such time and r to a room in R that is free at t. If there is no such time in $[L, U)$, t is set to a value above the interval, i.e., $t \geq U$; the value of r is then irrelevant. We assert without proof that $L \leq t$ is an invariant of the main loop in procedure *plan* given next.

```
procedure plan(P, R, L, U, t, r)

  t := L;
  while ¬com(t) ∧ t < U do
    t := ⟨max p : p ∈ P : p.next(t)⟩;
    t := ⟨min r : r ∈ R : r.next(t)⟩
  enddo ;
  {(L ≤ t) ∧ (com(t) ∨ t ≥ U)}

  if t < U then {com(t) ∧ L ≤ t < U}

    {reserve the professors in P at t}
    for p ∈ P do p.reserve(t) endfor ;

    {find a room in R and reserve it at t}
    r := ⟨min y : y ∈ R ∧ t = y.next(t) : y⟩;
    r.reserve(t)

  endif
end {plan}
```

1.2.3 Correctness and performance of plan

There are two ways to look at the correctness question: (1) *plan* is correct if none of the calendars (of the professors or the rooms) is changed during its execution by another program, and (2) *plan* is correct even when the calendars are changed during its execution. The first proposition, sequential correctness, is considerably easier to establish. For the current discussion, sequential correctness is not the central issue. There are well-known methods to establish such results; we refer the reader to sections 5.5.2 and 6.5.2 of this book for a thorough treatment of a variation of this problem. [3]

The second problem listed, correctness under concurrent execution, is very hard. Procedure *plan* may not work correctly if the calendar for some member of P or R is changed during its execution. In particular, concurrent executions of two instances of *plan* may reserve a room (or a professor) for two meetings simultaneously.

The problem is eliminated if each instance of *plan* gains exclusive access to the shared data, by explicitly locking the calendars of the members of P

[3] Correctness arguments can be based on the following facts: (1) any common meeting time in the interval $[L, U)$ is at least t (therefore, if *plan* returns such a time, t is the earliest common meeting time), and (2) if $\neg com(t) \wedge t < U$ holds, t will be increased eventually (therefore, either a common meeting time will be found or $t \geq U$ will hold).

and R before it commences execution. A more sophisticated strategy is to employ two-phase locking [20, 67]: all locks are acquired before any unlocking. Additionally, if the locks are acquired in a specific order, deadlock can be avoided. The programmer can introduce explicit locks into the code, or a compiler can insert them. Another possible protocol is as follows: each professor or room tentatively commits to a time whenever *next* is invoked and a commitment becomes permanent when *reserve* is invoked.

To execute several instances of *plan* concurrently, we can also exploit some of the properties of the program. For instance, if two instances of *plan* have disjoint sets of professors and disjoint sets of rooms or disjoint intervals $[L, U)$, their executions are non-interfering, and they can be executed concurrently. A more sophisticated scheme is to run exactly one iteration of the loop in each invocation of *plan*; if the iteration finds a common meeting time, then the rest of the procedure is executed to reserve the room and the professors and inform the caller; if no such time is found and $t < U$ after an iteration, then the call is rejected, i.e., the caller is asked to retry the call in the future. Thus, each call of *plan* locks the required data for only one iteration. Successive calls to *plan* may start with different calendars, and the requirement of the *earliest* meeting time may have to be replaced with *any* meeting time. However, such strategies are problem dependent; we cannot expect a program analyzer to deduce program properties and implement such strategies automatically.

1.3 Issues in Multiprogram Design

1.3.1 Concurrency is not a primary issue in design

We espouse the thesis that programmers should be concerned primarily with the problems they are solving and only secondarily with the implementation issues, such as concurrency. We have advocated this thesis for a number of years and demonstrated it in a number of examples in [32]. We continue to advocate that explicit concurrency considerations do not belong in program design, at least not in the early stages. A concurrent program should be designed as if each component in it will be executed in isolation; all other programs in the universe are suspended in favor of the executing component, and all state changes are attributable to this component alone.

The immediate consequence of this suggestion is that concurrent programming is now a vastly simpler task. Unfortunately, it is also a vastly impractical task because of severe degradation in performance. We examine these two issues next —correctness in this section and performance in section 1.3.3.

As we argued in section 1.2.3, correctness is much easier to establish if each component of a program is executed in isolation. In this book, the

unit of uninterrupted execution, called an *action*, is a sequential program.[4] If several actions have to be executed, they are executed in arbitrary, but serial, order; i.e., their internal steps are never interleaved. Thus, execution of an action completes before another is started.

Correctness of an individual action is established using traditional theories. An action is specified by a pair of predicates, its pre-condition and post-condition, and its correctness criterion is as follows: starting in a state where the pre-condition holds, execution of the action terminates in a state where the post-condition holds. (Termination is discussed later.) This aspect of programming and proof theory is in the domain of sequential programming, and we have little to say about it in this book. We are concerned largely with how to compose programs from objects and objects from procedures. We develop notations, methodology, and logic for designs of such programs. Correctness of a program can be deduced from the specifications of its constituent actions using some flavor of temporal logic [32, 118, 127, 128, 138, 139]; we develop an enhanced version of UNITY logic [32] in this book.

The constraint on executions of actions —execution of an action completes before another is started— has the consequence that "waiting" is now a meaningless concept. Since an action is executed alone, it cannot wait for another action to establish a condition for continuation of its execution. A process may wait neither to receive data along its input channel nor for a resource that it has requested to be granted; queuing up for a semaphore is a fruitless activity. Rendezvous-based communication that requires simultaneous participations of a sender and a receiver is outside our programming model. Our actions are all wait-free. Further, if an action is executed forever, it prevents execution of every other action. Therefore, execution of each action must be guaranteed to terminate (when started in an appropriate state). Termination guarantee is part of sequential correctness and is an obligation on the programmer. Our concern, therefore, is to develop a theory of programs consisting of wait-free, terminating actions.

1.3.2 Structuring through objects, not processes

The unit of abstraction in a typical concurrent program is a *process*. Processes are executed autonomously and concurrently, and they communicate with each other either through global shared variables or messages. Our model —a program is a set of wait-free, terminating actions— admits a different style of structuring, consisting of objects, and process communication is replaced by method call.

---

[4] An action can be a parallel program as long as its semantics can be specified by a pre-condition and a post-condition.

A program consists of a set of objects. Each object includes a set of *procedures*, where a procedure is either a method or an action. Actions and methods are similar; the only difference is that an action is executed autonomously, while a method is executed when it is called, as *p.next* and *r.next* are executed by being called from *plan*. The rule for action execution obeys a weak fairness condition: each action is executed infinitely often. (Therefore, a program execution is nonterminating, though each component action execution terminates.) Execution of a procedure —action or method— is strictly sequential: if a procedure calls a method of another object, the caller is suspended and resumes only on completion of the called method. Recall that completion of each method is guaranteed.

For the meeting planning problem, imagine that each *committee* of professors is represented by an object; this object may include an action that is executed periodically, say, at the start of each workweek to plan a meeting for that week. Procedure *plan* is a method that belongs to another object. Also, each professor and room is a separate object that includes the methods *next* and *reserve*. Execution of the action in *committee* initiates a call to *plan*, with professors in that committee and a set of appropriate rooms as arguments. Execution of *plan* calls on methods *next* and *reserve* of professor and room objects, as shown earlier. On completion of its execution, *plan* returns control to the calling action in *committee*. That action may then inform the members of the meeting time and the room (or that no meeting can be planned for that week).

Observe that there is no need to explicitly lock or unlock the calendars of the professors and rooms, because at most one instance of *plan* is executing at any moment. The program can be studied entirely as a sequential program, because concurrency aspects have been excluded during program design.

1.3.3 Implementation for efficient execution

The suggested execution strategy of one action execution at a time is only an illusion. The strategy makes it easier to design and understand programs, but it is totally impractical since it does not permit any concurrent execution; no two sites in the universe can have programs executing simultaneously.[5] What we want, ideally, is for the actions of a program to be executed concurrently for performance reasons, yet for humans to understand the program as if the actions are executed sequentially.

Two actions that are completely independent —i.e., no object is accessed or modified by both— can be executed simultaneously without causing interference. The notion of independence can be refined to allow concurrent

[5]Purists may argue that simultaneity is a meaningless concept in an Einsteinian universe.

executions of actions if their executions have the same effect as their serial executions in some order. In chapter 10, we define a binary relation, called *compatibility*, over the procedures and show that concurrent executions of compatible actions are equivalent to some serial executions of these actions.

Operations P and V on general semaphores are compatible and so are *put* and *get* over unbounded first-in–first-out channels. That is, whenever a call on *get* can be accepted, an execution of *put* before or after *get* has the same effect on the program state. However, operations *read* and *write* on a shared file are not compatible, as would be expected; the outcome of a *read* may depend on whether a *write* precedes or follows it. For the planning problem, *p.next* and *q.next* are compatible for all professors and rooms p and q (including $p = q$). However, *p.next* and *p.reserve* are not compatible because executing them in different order may yield different outcomes. Therefore, two invocations of *plan* cannot be executed concurrently if one may possibly call *p.next* and the other *p.reserve*.

Programmers have been successful in writing concurrent programs because, we believe, most pairs of actions are compatible. A scheduler can be employed to ensure that only compatible actions are executed concurrently; see an implementation in chapter 11. The programmer need only specify the pairs of methods in each object that are compatible; an efficient algorithm determines compatibility for all pairs of procedures given this information. The programmer's specification may be incomplete; if no pairs are specified to be compatible, the program is still executed correctly but with a reduced amount of concurrency. The scheduler in chapter 11 effectively simulates acquisition and release of locks. The scheduler can be distributed. Other implementation schemes, inspired by database commit protocols, can also be developed.

1.3.4 Transformational and reactive procedures

What happens when a procedure calls a method to request a resource and the resource is unavailable, such as attempting to receive a message from a channel that is empty? The called method can return an exception code to denote that it cannot be executed successfully. However, this type of interaction is common enough in concurrent programming that we distinguish between methods that always *accept* calls (execute their codes and, possibly, return some values) and those that may *reject* a call (to denote that the method cannot be executed in the present state). The former are called *total methods* and the latter *partial methods*. This distinction plays a central role in the programming model as well as in the development of the theory of concurrent execution.

A procedure in traditional sequential programming —to sort an array of integers, for instance— is a total method in our model. A procedure such as a P operation on a semaphore or a *get* operation on a channel is a partial method, because P and *get* can cause a caller to wait. Since

our model does not admit waiting, partial methods reject a call whenever completion cannot be guaranteed. In fact, a rejection should happen as early as possible in the execution of an action. In our model, rejection takes place before any change in the caller's state, and rejection itself does not affect the caller's state. Therefore, the caller is oblivious to rejection. In database terminology, rejection is "abort", and abort, in general, requires a rollback of the system to a valid state. However, our model avoids this problem because a call is rejected *before* causing any state change that requires rollback.

A rejection represents a transient condition, whereas acceptance represents a stable condition. In traditional concurrent programming, if a process polls its incoming channel and finds it empty, it cannot assert that it is empty (and, hence, start a computation based on channel emptiness) because the condition may be falsified even before the start of the computation.

A total procedure represents a *transformational* program; a partial procedure, a *reactive* program, in the terminology of Manna and Pnueli [127]. We exploit the distinction between total and partial procedures to get a weaker definition of compatibility (i.e., more pairs of actions are compatible —hence, more pairs can be executed concurrently— than would be possible if all methods were regarded as total). See section 3.4.1 for a longer discussion on partial and total procedures.

1.4 Concluding Remarks

Most process control systems —e.g., telephony, avionics— are conveniently represented using actions. Even an operating system can be structured in this manner. Typical actions in an operating system may be for garbage collection, response to a device failure, and allocation of resources in response to a request. A process control system includes actions that receive and process data from external sources, update internal data structures, and detect dangerous operating conditions. Each of these actions may involve a large amount of computation, but at the level of program design it makes sense to regard each action as a unit and design a larger system based on the units.

Programming of individual actions is a much-studied subject in the arena of sequential programming. This book contributes little to that effort. The emphasis in this book is on the *compositions* of actions and objects. Composition is fundamental for designs of complex software systems. Our work addresses some of the issues in program composition, including specifications of interfaces, predictions of system properties from the component properties, and design principles for "safe" compositions of subsystems.

A programming model is incomplete without an appropriate theory to aid its user in the analysis of programs. This is particularly true for concurrent programs because they tend to be harder. An action is often designed by assuming that the starting state, i.e., its pre-condition, satisfies some invariant. The obligation of the action is to reestablish the invariant as a post-condition. Additionally, establishment of progress properties, such as that execution of each action achieves a certain goal —planning a meeting, for instance— requires a theory that is more general than the study of invariants. We propose such a theory in this book.

1.5 Bibliographic Notes

The programming model that most closely resembles the approach presented here is transaction processing. There is a vast amount of literature on that subject; we refer the reader to Gray and Reuter [78] for a comprehensive survey. Bernstein and Lewis [19] contains a thorough treatment of concurrency issues in database systems. See Broy [26] for another approach to designs of distributed applications. Feijen and van Gasteren [69] have developed a beautiful approach, based on the classic work of Owicki and Gries [145], for designs of multiprograms, and they illustrate the approach convincingly on a large number of examples. It is yet to be seen if their work will scale up for larger problems. Jackson [95] discusses a number of thought-provoking issues in specification and programming methodology.

2
Action Systems

In chapter 1 we suggested that a program be structured as a set of objects. Each object consists of actions and/or methods, where the actions are executed autonomously (following a specific execution rule) and the methods are executed when they are called. In this chapter, we consider a simpler version of this model; we eliminate the methods altogether, retaining only actions. The immediate consequence of this decision is that the objects can no longer communicate through procedure calls; we require the objects to communicate via shared variables. Actions from different objects can read/write into these variables. However, at most one action is executed at any time, so there is no possibility of concurrent write into a variable.

This is an appropriate model for programs where communications among components play a minor role; computations of a single component are of the primary interest. We have chosen to study this simpler model —called *action systems*— because many of the basic concepts of the general model can be explained within it. The simpler model suffices for many problems; we can express the solution to a problem as an action system and study its properties employing a simple logic, which we develop in chapters 5 to 9. We describe the general programming model in chapter 3 and a logic for it in chapter 12.

of a program are restricted by the following fairness condition: *each action is executed infinitely often in each execution.*

It may seem that an infinite execution is meaningless if the computation is guaranteed to terminate. A terminating computation continues to execute its actions, but no action execution has any effect; therefore, the final state repeats forever. The execution rule defines a logical view of the execution; in an implementation, once it is detected that a final state has been reached, the execution may be stopped and the resources released for other tasks. However, the logical view is convenient for developing a uniform treatment of terminating and nonterminating computations.

Variable types The basic types used for variables in this book are **integer**, **boolean**, and **nat** (for natural, i.e., non-negative integers). Enumerated type with values $\{a, b, c, d\}$, for instance, is written as **enum** $\{a, b, c, d\}$. We simply write **type** to denote a polymorphic type, when type information has no relevance to the discussion. The structured types used are **record**, **set**, **bag**, **array**, and **seq** (for sequence). For a structured variable the type of its elements is also specified, and for each array its bounds. We write $\langle\ \rangle$ for an empty sequence, and $\emptyset$ for both empty set and empty bag. □

2.3 Properties of Action Systems

A thorough treatment of program properties is given in chapters 5 and 6. Here, we describe two of the main concepts —*invariant* and *fixed point*— that are necessary for understanding the examples in this chapter. Progress properties —that a program eventually reaches a desired state— are described in detail in chapter 6; for the moment, we rely on the reader's intuition to establish progress properties.

2.3.1 Invariant

An invariant is a predicate that is initially *true* and is preserved by execution of each action. Therefore, an invariant is always *true* during an execution. (The states reached *during* an execution are the initial state and the state following the execution of each action. The states that are reached *during* execution of an action are invisible; we can observe the states only before and on completion of each action execution.) Formally, predicate p is an invariant if both of the following conditions hold.

initial condition $\Rightarrow$ p
for each action of the form $g \rightarrow s$, $\{p \wedge g\}\ s\ \{p\}$

Here, $\{p \wedge g\}\ s\ \{p\}$ denotes that any execution of s started in a state that satisfies $p \wedge g$ terminates in a state that satisfies p; see appendix A.4.1 for details about this notation.

As an example, consider a program that consists of the following box only.

```
box small
  integer x, y = 0, 0;

    x < y  →  x := x + 1
  [] y := max(x, y) + 1
end {small}
```

We claim that $x \leq y$ is an invariant for this program. We have

initially $x = 0 \ \wedge \ y = 0$

which implies $x \leq y$. We can show that

$$\begin{array}{lll} \{x \leq y \ \wedge \ x < y\} & x := x + 1 & \{x \leq y\} \\ \{x \leq y\} & y := max(x, y) + 1 & \{x \leq y\} \end{array}$$

The notion of invariant is perhaps the most important foundational concept in this book. It is essential for writing specifications and designing programs.

2.3.2 fixed point

A fixed point of a program is a state that remains unchanged by execution of any action. Therefore, once a fixed point is reached, further execution of the program has no effect. The set of all fixed points is described by a predicate called *FP*.

It is possible to compute *FP* from the code of a program provided that we know the states left unchanged by each action. Consider a program whose action i is of the form $g_i \ \rightarrow \ s_i$. Let predicate b_i hold in exactly those states where the execution of s_i has no effect. Then

$$FP \ \equiv \ \langle \forall i :: \ g_i \Rightarrow b_i \rangle$$

Observe that *FP* holds in any state where all g_is are *false*.

It is easy to compute b_i if s_i is an assignment statement. For the assignment statement $x := e$ the corresponding predicate is $x = e$. That is, execution of $x := e$ has no effect exactly when $x = e$ holds prior to the execution. This observation may easily be extended to sequences of assignments and conditional statements; see section 5.3.2 for details. For program *small* of section 2.3.1, we compute[1]

[1]See appendix A.2.1 for an explanation of the proof format used here.

$$
\begin{array}{ll}
 & FP \\
\equiv & \{\text{from the definition of } FP\} \\
 & (x < y \Rightarrow x = x + 1) \ \wedge \ (y = max(x, y) + 1) \\
\equiv & \{\text{arithmetic and predicate calculus}\} \\
 & (x \geq y) \wedge (\mathit{false}) \\
\equiv & \{\text{predicate calculus}\} \\
 & \mathit{false}
\end{array}
$$

That is, each state of *small* can potentially be changed.

There is no direct method for computing the *FP* if the command portion of an action contains loops.

Most of the systems we consider in this book are never expected to reach a fixed point; they should run forever, so their *FP* should be *false*. In many cases, though, a box may reach a fixed point, but then a change in a shared variable by some other box may cause its *FP* to become *false*, and enable some of its actions to be executed effectively.

2.4 Examples

2.4.1 Finite state machine

Finite state machines are conveniently represented by action systems: the machine state can be encoded in a variable, and each state transition is an action. Alternatively, it may be possible to define a set of variables where the variable values encode the states and each transition affects only a small number of variables.

We show two different representations of a finite state machine that accepts binary strings that have an even number of zeroes and an odd number of ones. A pictorial representation of the machine is given in Fig. 2.1. In this figure, the initial state is a and state c is the only accepting state.

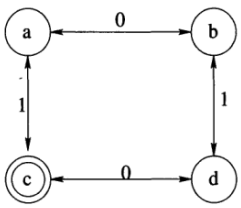


Figure 2.1: Finite state machine accepting even number of 0's and odd 1's

A box *FSM1* that represents this finite state machine follows. Variable *state* assumes one of the values a, b, c, and d. Variable x holds the next binary digit to be scanned. Some external box E stores a value into x after

FSM1 has scanned the digit; E is usually called the *environment* of *FSM1*. The following protocol is used by E and *FSM1* to read/write into x. The value of x is ϕ when there is no value to be scanned; in this case box E may store a binary digit in x. Box *FSM1* reads a value from x if $x \neq \phi$ and then it sets x to ϕ.

---

box *FSM1*
  enum $\{a, b, c, d\}$ $state = a$;
  enum $\{0, 1, \phi\}$ x;

  $x = 0 \rightarrow$ **if** $state = a$ **then** $state := b$
    elseif $state = b$ **then** $state := a$
    elseif $state = c$ **then** $state := d$
    else $\{state = d\}$ $state := c$
    endif ; $x := \phi$

  $[\!] \; x = 1 \rightarrow$ **if** $state = a$ **then** $state := c$
    elseif $state = b$ **then** $state := d$
    elseif $state = c$ **then** $state := a$
    else $\{state = d\}$ $state := b$
    endif ; $x := \phi$
end $\{FSM1\}$

Box *FSM1* reaches a fixed point when $x \neq 0 \wedge x \neq 1$, i.e., $x = \phi$. Then *FSM1* is merely waiting for input from its environment.

In the following box, we encode the state by two boolean variables $p0$ and $p1$, where $p0$ is *true* iff the number of scanned 0's is even; $p1$ is similarly defined. Thus, states a, b, c, d are encoded by the following values of $p0, p1$, respectively: $(true, true)$, $(false, true)$, $(true, false)$, $(false, false)$. Note that the resulting box is considerably simpler because of the choice of variables that represent the states.

---

box *FSM2*
  boolean $p0, p1 = true, true$;
  enum$\{0, 1, \phi\}$ x;

  $x = 0 \rightarrow$ $p0 := \neg p0$; $x := \phi$
  $[\!] \; x = 1 \rightarrow$ $p1 := \neg p1$; $x := \phi$
end $\{FSM2\}$

Let $n0$ and $n1$ denote, respectively, the number of 0's and 1's scanned. Variables $n0$ and $n1$ are *auxiliary* variables that can be introduced into

FSM2: initially, both of these variables are 0; $n0$ is incremented in the first action and $n1$ in the second. It can be shown that

invariant $p0 \equiv even(n0)$
invariant $p1 \equiv even(n1)$

2.4.2 Odometer

We consider a three-digit odometer whose state is described by the values of the variables $d0$, $d1$, and $d2$ ($d0$ is the least significant and $d2$ the most significant digit). An external process, the environment of the odometer, sets variable $c0$ to *true* to signify that the odometer should be incremented. The odometer is incremented eventually if $c0$ remains *true*, and then $c0$ is set to *false* (to denote that the incrementation has been completed).

In the first design, we have a single action that increments the odometer when $c0$ is found to be *true*.

---

box *Odometer1*
  enum(0..9) $d0, d1, d2 = 0, 0, 0$;
  boolean $c0$;

  $c0 \rightarrow c0 := false$;
    $d0 := (d0 + 1)$ *mod* 10;
    if $d0 = 0$ **then** $d1 := (d1 + 1)$ *mod* 10;
      if $d1 = 0$ **then** $d2 := (d2 + 1)$ *mod* 10 **endif**
    endif
end {*Odometer1*}

Observe that if $c0$ becomes *true*, from the fairness condition, the odometer will be incremented and $c0$ set to *false*.

There is a deficiency in our modeling of a physical odometer as an action system. We cannot guarantee that the odometer will be incremented within a very short time of $c0$ being set to *true*; the guarantee that the odometer is incremented eventually may have little value in practice if several miles elapse before an incrementation. We discuss this issue in some detail in chapter 6.

A note on the notation We have not distinguished variable $c0$ from variables $d0$, $d1$, and $d2$ syntactically, even though the latter variables are local to the box (i.e., they cannot be changed by an external action) whereas $c0$ can be changed by an external action. We introduce a syntactic distinction in section 8.2.1. □

$$
\begin{array}{ll}
 & FP \\
\equiv & \{\text{from the definition of } FP\} \\
 & \langle (m > n) \;\Rightarrow\; (m, n = m - n, n) \rangle \;\wedge \\
 & \langle (n > m) \;\Rightarrow\; (m, n = m, n - m) \rangle \\
\equiv & \{\text{Simplify}\} \\
 & \langle (m > n) \Rightarrow (n = 0) \rangle \;\wedge\; \langle (n > m) \Rightarrow (m = 0) \rangle
\end{array}
$$

Any fixed point reached by the box satisfies the invariant and this FP; hence, at a reachable fixed point

$$
\begin{array}{ll}
 & P \;\wedge\; \langle (m > n) \Rightarrow (n = 0) \rangle \;\wedge\; \langle (n > m) \Rightarrow (m = 0) \rangle \\
\Rightarrow & \{P \;\Rightarrow\; n > 0. \text{ And } m > n \;\Rightarrow\; n = 0. \text{ So } m \leq n. \\
 & \quad \text{Similarly, } n \leq m\} \\
 & m \leq n \;\wedge\; n \leq m \;\wedge\; gcd(m, n) = gcd(M, N) \\
\Rightarrow & \{\text{arithmetic}\} \\
 & m = n \;\wedge\; gcd(m, n) = gcd(M, N) \\
\Rightarrow & \{gcd(x, x) = x, \text{ for any positive integer } x\} \\
 & m = gcd(M, N)
\end{array}
$$

The remaining proof obligation is that every execution of GCD eventually reaches a fixed point. This result does not follow from anything we have proved so far: if we replace $m - n$ with $m + n$ and $n - m$ with $n + m$, all the proof steps remain valid, yet the box will never reach a fixed point. Since we have not developed a theory of progress, we provide an operational argument to justify that a fixed point will be reached. Observe that if $m \neq n$, execution of one of the actions changes m or n, thus decreasing $m + n$, whereas the other action has no effect. From the fairness rule that each action is eventually executed, we conclude that $m + n$ will be decreased eventually if $m \neq n$. Since both m and n are always positive (see the invariant), $m + n$ can be decreased a finite number of times only. Hence, within finite time $m = n$, and this implies FP.

2.4.4 Merging sorted sequences

This example demonstrates that message-communicating processes may be represented easily as action systems. We design a box that merges the data received along three input channels. Each channel carries an increasing sequence of positive integers; the output of the box is an increasing sequence that includes all (and only) the received values, and this sequence is sent along an output channel. Since the output sequence is increasing, no value appears more than once in the output channel, even though the same value may appear in different input channels. This box is used as part of a larger example in section 4.5.

The shared variables in this example are channels. A channel is an unbounded sequence; an empty channel is denoted by the empty sequence, $\langle\rangle$. Sending value x along channel c has the same effect as

$$c := c +\!\!+ x$$

where $+\!\!+$ is the concatenation operator. Receiving a value from c into v is effected by

$$c \neq \langle\rangle \;\rightarrow\; v, c := c.head, c.tail$$

In this example, some external box appends values to the input channels, and box *Merge*, shown below, removes values from these channels. Dually, *Merge* appends values to the output channel, and some external box receives those values. The protocol shown here guarantees that the channels are first-in–first-out (fifo).

The algorithm used in *Merge* is as follows. The input channels are called f, g, and h, and the output channel, *out*. Each input channel has an integer variable associated with it —vf, vg, and vh with f, g, and h, respectively— that holds the last value read from the channel that has not yet been output; in case all values read from a channel have been output, the corresponding variable value is 0 (recall that the channels carry only positive integers). A value is read from channel f and stored in vf provided that $vf = 0$ and the channel is nonempty; similarly for the other channels. A value is output only if vf, vg, and vh are all nonzero; in that case, the smallest of these values is output, and vf, vg, vh are appropriately modified.

---

box *Merge*
  seq $f, g, h, out;$
  integer $vf, vg, vh = 0, 0, 0;$
  integer $m;$

$$\begin{array}{ll} & vf = 0 \;\wedge\; f \neq \langle\rangle \;\rightarrow\; vf,\ f \;:=\; f.head,\ f.tail \\ [\!] & vg = 0 \;\wedge\; g \neq \langle\rangle \;\rightarrow\; vg,\ g \;:=\; g.head,\ g.tail \\ [\!] & vh = 0 \;\wedge\; h \neq \langle\rangle \;\rightarrow\; vh,\ h \;:=\; h.head,\ h.tail \\ [\!] & vf \neq 0 \;\wedge\; vg \neq 0 \;\wedge\; vh \neq 0 \;\rightarrow \\ & \quad m := min(vf, vg, vh);\ out := out +\!\!+ m; \\ & \quad \textbf{if}\ m = vf\ \textbf{then}\ vf := 0\ \textbf{endif}; \\ & \quad \textbf{if}\ m = vg\ \textbf{then}\ vg := 0\ \textbf{endif}; \\ & \quad \textbf{if}\ m = vh\ \textbf{then}\ vh := 0\ \textbf{endif} \end{array}$$

end {*Merge*}

Box *Merge* expects a never-ending stream of values along each input channel. In case a channel carries a finite number of values, some of the values from the other channels may never be output (for instance, if f carries some values and g and h are permanently empty). In that case, each finite sequence should be terminated by a special end marker, say ∞, and the box should be modified to ignore that channel after receiving the special value.

The properties of *Merge* that are of interest are as follows.

1. Each value in *out* is from f, g, or h.

2. *out* is a strictly increasing sequence.

3. Each value from f, g, and h appears eventually in *out*.

The first two properties can be stated as invariants of *Merge* and the last one is a progress property.

2.4.5 Mutual exclusion

Mutual exclusion is a classic problem in concurrent computing. We treat the problem here not because of its intrinsic difficulty or its central place in concurrent computing but as an illustration of refinement in action systems.

Two or more processes each have a section of code called the *critical section*, and it is required that at most one process execute its critical section at any time. Therefore, if two processes attempt to execute their critical sections simultaneously, then one of them will be forced to wait at least until the other has completed execution of its critical section. Additionally, a reasonable progress requirement is that some process eventually executes its critical section if there are processes waiting to enter their critical sections. A stronger progress requirement is that every waiting process eventually be allowed to enter its critical section.

The *Merge* example of section 2.4.4 is part of a *loosely coupled* system, where the components —boxes that write into the input channels of *Merge* and read from its output channel, and the *Merge* box itself— can be developed and understood without detailed understanding of the other components. These components interact only through the shared channels, and such interactions are easy to understand. The thesis in this book is that all large programs should be loosely coupled. In contrast to *Merge*, a solution to the mutual exclusion problem is usually *tightly coupled*; such a solution is difficult to understand by examining the code of each process in isolation. The shared variables are manipulated in an intricate manner, and it is preferable to study the program, consisting of all its components, in its entirety.

In this section, we develop a mutual exclusion algorithm due to Peterson [151]. We start with a high-level solution that is loosely coupled. Next, we refine this solution, implementing a complex shared data structure using elementary data structures. Ultimately, we represent Peterson's solution as a single action system. To show the power of refinement, we derive a second mutual exclusion algorithm from the same high-level program.

From multiple assignments to single assignments The preceding algorithm is almost identical to Peterson's two-process mutual exclusion algorithm. The remaining step is to decouple the assignments to u and *turn* in process u (and similarly v and *turn* in process v). We can show that (see Misra [134, note 13]) it is safe to replace

$u, turn := true, true$
by
$u := true;\ turn := true$ □

Note Switching the order of the two assignments for either process makes the program incorrect. To see this, suppose that processes u and v have the following codes.

process u:: $u := true;\ turn := true$
process v:: $turn := false;\ v := true$

Consider an execution in which process u sets u to *true*, process v sets *turn* to *false*, and u then sets *turn* to *true*. Now u enters its critical section ($\neg v$ holds); then, process v sets v to *true* and enters its critical section (because *turn* holds), thus violating mutual exclusion. □

Peterson's algorithm as an action system

It is easy to translate the two-process mutual exclusion program into an action system. First, we rewrite the program using two explicit program counters —m for process u and n for process v— that take on integer values between 0 and 3.

---

program *MutualExclusionRefined1*
  boolean $u, v = false, false;$
  integer $m, n = 0, 0;$

process u	process v
loop	**loop**
noncritical section;	noncritical section;
$u, m := true, 1;\ turn, m := true, 2;$	$v, n := true, 1,\ turn, n := false, 2;$
$\neg v \vee \neg turn \rightarrow skip;$	$\neg u \vee turn \rightarrow skip;$
{enter critical section} $m := 3;$	{enter critical section} $n := 3;$
critical section;	critical section;
$u, m := false, 0$	$v, n := false, 0$
end	**end**

end {*MutualExclusionRefined1*}

We translate this program to the action system shown below. In the translation, we introduce predicates $u.h$ and $v.h$, which are controlled by external boxes. Predicate $u.h$ is set to *true* to denote that process u is waiting to enter its critical section, and it is set to *false* while process u is in its critical section; $v.h$ is manipulated similarly.

The fact that every critical section is eventually completed is simulated by setting m to 0 sometime after it becomes 3 (similarly for n).

---

program *mutex*
  boolean $u, v = \mathit{false}, \mathit{false}$;
  integer $m, n = 0, 0$;

    {process u's box}
  $u.h \ \wedge \ m = 0 \ \rightarrow \ u, m := \mathit{true}, 1$
$[\!]$ $m = 1 \ \rightarrow \ \mathit{turn}, m := \mathit{true}, 2$
$[\!]$ $m = 2 \ \wedge \ (\neg v \ \vee \ \neg \mathit{turn}) \ \rightarrow \ m := 3$
$[\!]$ $m = 3 \ \rightarrow \ u, m := \mathit{false}, 0$

    {process v's box}
$[\!]$ $v.h \ \wedge \ n = 0 \ \rightarrow \ v, n := \mathit{true}, 1$
$[\!]$ $n = 1 \ \rightarrow \ \mathit{turn}, n := \mathit{false}, 2$
$[\!]$ $n = 2 \ \wedge \ (\neg u \ \vee \ \mathit{turn}) \ \rightarrow \ n := 3$
$[\!]$ $n = 3 \ \rightarrow \ v, n := \mathit{false}, 0$
end{*mutex*}

Proof of mutual exclusion

We constructed program *mutex* through a series of transformations starting from the program that used a shared queue. Since *mutex* is a correct refinement of a correct mutual exclusion algorithm it also enforces mutual exclusion. That is, m and n cannot both be 3 simultaneously:

invariant $\neg(m = 3 \ \wedge \ n = 3)$

This fact cannot be proved directly from the program text; we prove invariants (I1) and (I2), given below, from which this fact can be deduced.

invariant $\langle m \neq 0 \ \equiv \ u \rangle \ \wedge \ \langle (m = 3) \ \Rightarrow \ (\neg v \vee \neg \mathit{turn}) \rangle$ (I1)
invariant $\langle n \neq 0 \ \equiv \ v \rangle \ \wedge \ \langle (n = 3) \ \Rightarrow \ (\neg u \vee \mathit{turn}) \rangle$ (I2)

Invariants (I1) and (I2) can be proved by showing that they hold initially and that every action preserves the truth of each of these predicates. The proof is straightforward, and we leave it to the reader. Given (I1) and (I2), we conclude from their conjunction that both processes cannot be in their critical sections simultaneously, as follows.

```
program MutualExclusionRefined2
  boolean u, v = false, false;
  boolean p;

process u                          process v

loop                               loop
  noncritical section;               noncritical section;
  p, u := v, true;                   p, v := ¬u, true;
  ¬p → skip;                         p → skip;
  critical section;                  critical section;
  p, u := true, false                p, v := false, false
end                                end

end {MutualExclusionRefined2}
```

This program has the advantage over Peterson's that exactly one boolean variable has to be checked in the guarded command. Unfortunately, the program requires assignments of the form $p := v$ and $p := \neg u$, naming shared variables on both sides of an assignment, which are difficult to implement as atomic actions.

The multiple assignment statements can be replaced by the following sequences of single assignments; see [134, note 13] and also see the note on page 28.

$$p, u := v, true \quad \text{by} \quad u := true;\ p := v$$
$$p, v := \neg u, true \quad \text{by} \quad v := true;\ p := \neg u$$

2.4.6 Shortest path

Dijkstra's shortest path algorithm [56] has by now become a classic (the cited paper is officially designated "classic" by the Citation Index Service). Typical descriptions (and derivations) of this algorithm start by postulating that the shortest paths be enumerated in the order of increasing distances from the source. In this section, we present a derivation that is quite different in character. We view the problem as the computation of a "greatest solution" of a set of equations. We prescribe an action system whose implementation results in Dijkstra's algorithm.

The bulk of the work in our derivation is in designing the appropriate heuristics that guarantee termination (i.e., reaching a fixed point); this is in contrast to traditional derivations, where most of the effort is directed toward postulating and maintaining the appropriate invariant.

Index