EVENTUAL DETERMINISM: USING PROBABILISTIC MEANS TO ACHIEVE DETERMINISTIC ENDS

Josyula R. Rao

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-90-08 April 1990

Eventual Determinism: Using Probabilistic Means to Achieve Deterministic Ends

Josyula R. Rao*
Department of Computer Sciences
The University of Texas at Austin

January 31, 1990

Abstract

We introduce a new paradigm for the design of parallel algorithms called *eventual determinism*. In an *eventually-determinizing* algorithm, all processes execute identical programs from identical starting states. A program has two parts (alternatively, called *modes*) – *probabilistic* and *deterministic*. A process begins execution in the probabilistic mode and eventually (with probability one) switches to a deterministic mode. The decision to switch is taken independently by each process. Since different processes can execute in different modes, it is required that the mode of a process be transparent to its environment. Thus, determinacy pervades the system.

Eventually-determinizing algorithms combine the advantages of probabilistic and deterministic algorithms. We illustrate the design of such an algorithm for a problem of conflict-resolution for distributed systems. We construct an algorithm for a ring of dining philosophers by combining a modified version of the probabilistic Lehman-Rabin's Free Philosopher algorithm with the deterministic Chandy-Misra algorithm. The system is proved to be starvation-free using a new proof-system for reasoning about probabilistic algorithms proposed by the author. The proof technique is novel in two ways : firstly, it allows the manipulation of probabilistic and deterministic properties within one unified framework. Secondly, existing proofs of component algorithms can be used along with a proof of correctness of their interaction to construct a proof of the algorithm as a whole.

*This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658-065 and by grant ONR 26-0679-4200 from the Office of Naval Research.

Contents

0 Introduction

0.1 Motivation

Ever since Michael Rabin's seminal paper on Probabilistic Algorithms [Rab76], several algorithms employing randomization have appeared in the literature [BGS88]. Often these algorithms are simpler and more efficient – in terms of space, time and communication complexity – than their deterministic[0] counterparts. It has also been recognized that for certain problems, especially in the areas of multi-processing and distributed computing, it is possible to construct a probabilistic solution where no deterministic one exists. One such problem is that of resolving symmetry in a parallel environment [Dij74, LR81].

[0] We use the term *deterministic* to mean *non-probabilistic*.

In a parallel system consisting of a multitude of processes, it is possible to customize each process with a special starting state or program. However as the multitude grows, the cost of customizing increases and symmetry becomes an interesting and desirable feature. For our purposes, a system of processes is said to be *symmetric*, if all processes have identical starting states and execute identical programs. It has been shown in [Dij74, LR81], that such a symmetric system can reach an asymmetric state only if the processes are allowed to make probabilistic transitions. It is not possible to break the symmetry of a parallel environment in a deterministic and deadlock-free manner.

Although probabilistic algorithms have several charms like simplicity, efficiency and tractability, trading determinacy for randomization has its price. The traditional notion of absolute correctness of deterministic algorithms has to be generalized to a notion of correctness with a quantitative probability. This means that some probabilistic programs may take an inordinate amount of time to execute. Complexity measures for such algorithms tend to be expected values rather than upper bounds. On the other hand, there is a large and growing body of literature of deterministic algorithms that have provably optimal upper bounds using varying complexity measures.

0.2 Contributions

To bridge the gap between the capabilities of probabilistic and deterministic algorithms and to harness the advantages of both, we propose a new paradigm – *eventual determinism*. An *eventually-determinizing system* of processes is symmetric and all processes execute an algorithm with two parts (alternatively called *modes*) – *probabilistic* and *deterministic*. Each process begins execution from the same starting state in the probabilistic mode. When the system reaches a legal starting state for the deterministic part, each process switches to the deterministic mode and remains in that mode thereafter. In many applications, the legal starting state for the deterministic algorithm is simply an asymmetric state. It is required that the decision to switch modes be made locally. Furthermore each process that begins execution, switches mode with probability one.

An *eventually-determinizing* algorithm should satisy the following criteria.

- *Symmetry* : All processes execute identical programs from identical starting states.
- *Locality* : Each process begins execution in a probabilistic mode and *independently* makes the decision to switch to a deterministic mode. Thus a process can execute in a deterministic mode while another process is executing in the probabilistic mode. This switch is made by each process with probability one.
- *Transparency* : The mode of a process is transparent to the processes it synchronizes with.
- *Uniformity* : Since probabilistic and deterministic processes co-exist in the system and transparency is required, a uniform and clean interface is needed between the probabilistic and deterministic modes.
- *Eventual Determinacy* : The system of processes gradually becomes deterministic.

Note that in an eventually-determinizing algorithm, it is sufficient for the probabilistic mode to establish a legal starting state for the deterministic mode. Thus the specification of the probabilisitic mode can be much weaker than the specification of the algorithm. Also, existing proofs of the probabilistic and deterministic modes can be used along with a proof of correctness of their interaction to construct a proof of the eventually-determinizing algorithm.

In this paper, we illustrate the idea of eventual determinacy by a problem of fair conflict-resolution for distributed systems. We consider the classic dining philosophers problem and restrict attention to an arbitrary but finite number of philosophers seated around a circular table. The problem is well-known [Dij72] and will not be described here.

In [LR81], Lehman and Rabin describe a probabilistic scheme for ensuring deadlock-freedom for a symmetric system of dining philosophers arranged in a ring. Deadlock freedom guarantees that if at any time a philosopher is hungry then at a later time *some* philosopher eats. In [CM84], Chandy and Misra present a deterministic algorithm for ensuring starvation freedom for an arbitrary network of dining philosophers. The algorithm guarantees that if a philosopher is hungry, then after a finite delay it eats. In this algorithm, neighboring processes share a fork (called the *common* fork) and the fork can be either clean or dirty. The act of eating dirties the fork. A fork is cleaned by a process only when it is sent to the neighbor. A process has priority over its neighbor if and only if it has the common fork and the fork is clean or the common fork is with the neighbor and is dirty. The algorithm manipulates the precedence graph of processes which reflects this priority of a process with respect to its neighbors. One of the key invariants of the algorithm requires the precedence graph to be acyclic and this necessitates an asymmetric starting state. Unlike the Lehman-Rabin algorithm however, there is a bound on the number of message communications required of a hungry philosopher before it eats.

We describe an eventually-determinizing algorithm which combines these algorithms to reap the benefits of both. The Lehman-Rabin algorithm has been described in a manner suitable for a shared-memory architecture. We present a version of the algorithm suitable for interfacing with the Chandy-Misra algorithm and execution in a distributed environment. The rule for switching to a deterministic mode is simple : a philosopher can start executing the deterministic Chandy-Misra algorithm if it either receives a clean fork or begins eating.

All philosophers begin by executing the modified Lehman-Rabin algorithm in the same starting state. After a philosopher becomes hungry, since the Lehman-Rabin algorithm guarantees deadlock-freedom, some philosopher eats. This act of eating has two important effects. Firstly the philosopher becomes deterministic. More importantly, it causes the philosopher's priority to become lower than its neighbors. This causes the precedence graph for a ring configuration to become acyclic. The transition rules of the modified Lehman-Rabin and Chandy-Misra algorithms are such that this acyclicity condition cannot be violated. If a neighboring philosopher becomes hungry, it requests a fork from this deterministic philosopher and gets a clean fork, enabling it to transit to a deterministic state. Thus determinacy is contagious and spreads through the ring.

Reasoning about probabilistic algorithms is still an active field of research and there are very few proof systems that address the methodological problem of refining specifications and reasoning about the composition of probabilistic programs. By synthesising ideas from the theory of the weak predicate transformer [DS89], UNITY [CM88] and existing proof techniques for probabilistic algorithms [HSP83, Pnu83], the author has developed such a proof system [Rao90]. The proof system comprises a minimalistic computational model which captures the basic notions of asynchrony, synchrony and probabilistic choice. It provides a small set of versatile operators expressive enough to reason about safety and progress properties of probabilistic parallel programs. It also supports hierarchical and compositional development of such programs. Conditional properties allow proofs to be abstracted and reused. Eventually-determinizing algorithms provide an ideal testbed for such proof systems as they require abstraction and compositional reasoning about probabilistic and deterministic progress properties. The proof of the eventually-determinizing algorithm described above has been carried out using this proof-system.

The problem of eventual determinism was suggested by Jayadev Misra. Eliezer Levy [Lev88] was the first to apply it to the dining philosophers problem. Though our basic idea is the same as his, he makes stronger assumptions about the Lehman-Rabin algorithm than we do. Our development has been aided by the proof techniques for reasoning about composition of probabilistic programs.

The rest of the paper is organized as follows. After introducing the problem in Section 1, we describe the variables and notation of our solution in Section 2. The original Free Philosophers Algorithm of Lehman-Rabin and our modification of it are presented in Section 3. In Section 4, we give a brief description of the Chandy-Misra algorithm. The eventually-determinizing algorithm and a proof sketch of its correctness appear in Section 5. Finally, we conclude with ideas for future research.

1 The Symmetric Dining Philosophers Problem

Processes, called *philosophers* are placed at the vertices of a ring, with one philosopher at every vertex. A philosopher can be in one of three states : *thinking*, *hungry* or *eating*. Associated with each edge of the ring is a *fork*. A philosopher can eat only if it possesses the forks corresponding to both its incident edges. A thinking philosopher may become hungry. A hungry philosopher remains hungry until it gets its incident forks, when it begins to eat. On entering the eating state, a philosopher eats for a finite period and then transits to the thinking state. A philosopher may think for an arbitrary period of time.

The problem is to design an eventually determinizing algorithm that is starvation-free. That is, no philosopher remains hungry indefinitely.

2 Notation and Variable Declarations

In this section, we define the variables used by the eventually determinizing algorithm. The philosophers are assumed to be numbered from 0 through $N-1$. Associated with each philosopher u are the following variables.

- $u.dine$: This can take values t, h or e corresponding to *thinking*, *hungry* or *eating* respectively. For convenience, we abbreviate by a boolean variable $u.t$, the boolean predicate $(u.dine = t)$. The boolean variables $u.h$ and $u.e$ are similiarly defined.
- $u.mode$: This can take values p or d corresponding to *probabilistic* or *deterministic* respectively. For convenience, we abbreviate by a boolean variable $u.p$, the boolean predicate $(u.mode = p)$. The boolean variable $u.d$ can be similiarly defined.
- $draw.u$: This can take values $u-1$, $u+1$ or N. This variable is intended to contain the value of a random choice made by a process. It will be used in the Lehman-Rabin Algorithm.

An edge between philosophers u and v is denoted by (u, v) (or equivalently (v, u)). Two philosophers are said to be *neighbors* if and only if there is an edge between them. For convenience, the ring is assumed to be represented by a boolean matrix E, where $E[u, v]$ holds if and only if there is an edge between vertices u and v. Associated with each edge (u, v) of the ring, are the following variables.

- $fork[u,v]$: The *fork* is shared by neighboring philosophers u and v and can take values u or v indicating the identity of the philosopher in possession of the fork.
- $clean[u,v]$: This boolean variable is an attribute of the fork and indicates whether the corresponding fork is *clean* or *dirty*.
- $rf[u,v]$: The *request token* is shared by neighboring philosophers u and v and can take values u or v indicating the identity of the philosopher in possession of the request token.
- $rfclean[u,v]$: This boolean variable is an attribute of the request token and indicates whether the corresponding request token is *clean* or *dirty*.
- $rffirst[u,v]$: This boolean variable is an attribute of the request token.

For an edge (u,v), the reflection function $R(u,v)$ denotes the second neighbor of v.

3 The Lehman-Rabin Algorithm

In [LR81], Lehman and Rabin describe two probabilistic algorithms to solve the problems of deadlock-freedom (the Free Philosopher's Algorithm) and starvation-freedom (The Courteous Philosopher's Algorithm) for a finite number of dining philosophers arranged around a circular table. In the sequel, we shall be concerned only with the Free Philosopher's Algorithm.

We give an informal description of the original Free Philosopher's Algorithm. In the starting state, all forks are down and all philosophers are thinking. When a philosopher becomes hungry, it makes a random choice between its left and right neighbor. If the neighbor of choice is holding the common fork, the philosopher waits for the fork to be put down. Once the common fork is down, the philosopher picks it up and then inspects the status of the second fork it needs. If the second fork is down, the philosopher picks it up and starts eating. On leaving the eating state, it puts down both forks (in any order) and starts thinking. If the second fork is being used by its neighbor, it puts down the first fork and repeats the whole process by making another random choice.

It is important to note the subtle distinction between the first and second forks. While a philosopher waits for the first fork to be put down, it does not wait for the second one. Clearly, a fork being up or down is determined by whether the neighbor is using it or not. How are "up" and "down" implemented in a distributed environment ? How does a philosopher request a fork ? Under what conditions can a philosopher release a fork ? More importantly, when can a philosopher assume that a neighbor is using a fork and stop waiting ?

To tailor the Free Philosopher's algorithm to execute in a distributed environment, we reformulate it using the variables introduced in Section 2. The variable $draw.u$ reflects the random choice by philosopher u. It has the values N, $u-1$ or $u+1$ to denote repectively that it has not drawn, drawn the left neighbor or drawn the right neighbor respectively.

Having made a random choice, a philosopher uses the request tokens it possesses to requisition the forks it doesn't have. If philospher u is hungry and needs $fork[u,v]$, it sends request token $rf[u,v]$ to the neighbor v who is currently holding the fork. The $rffirst[u,v]$ attribute of the request token is set to inform philosopher v of the random choice of philosopher u. That is, $rffirst[u,v]$ is set to true if $(draw.u = v)$ and it is set to false if $(draw.u = R(v,u))$. This information helps philosopher v decide, whether an immediate response to philosopher u is necessary or not. A true value for $rffirst[u,v]$

indicates that philosopher u will wait for v to put the fork "down", whereas a false indicates it is enough to inform philosopher u that the fork is "in use".

To inform philosopher u that $fork[u, v]$ is "in use", philosopher v "dirties" the request token and sends it to philosopher u. That is, the request token $rf[u, v]$ is returned to u with a false value for the $rfclean[u, v]$ attribute. On receipt of a dirty request token, philosopher u stops waiting, cleans the request token and makes another random choice.

The starting state of the algorithm is the same for each philosopher and is given as follows.

- All philosophers are thinking.
- No philosopher has made a random choice.
- For a given edge, the fork and request token are held by different philosophers.
- All request tokens are clean.
- The $rffirst$ attribute of each request token is set to true.

The algorithm is described in the appendix using a UNITY based notation developed in [Rao90] The proof obligation of the distributed version of the Lehman-Rabin algorithm is twofold.

- *Mutual Exclusion* : Neighboring philosophers do not eat simultaneously.
- *Deadlock Freedom* : If a philosopher is hungry, then it is guaranteed, with probability one, that *some* philosopher will eat.

The proof has been formally worked out using a proof-system developed by the author. The proof is similiar to one given by Pnueli and Zuck [PZ86] for the original Free Philosopher's algorithm.

4 The Chandy-Misra Algorithm

In [CM84], Chandy and Misra present a deterministic solution to the dining philosphers problem. Although their solution is applicable to an arbitrary network of philosophers, we are interested in applying their solution to a ring of philosophers.

The key idea of the algorithm is to maintain asymmetry as an invariant. This necessitates an asymmetric starting state. In the algorithm, asymmetry is represented by the *acyclicity* of a precedence graph of philosophers. This precedence graph is identical to the interconnection graph of the philosophers (a ring, in our case) except that each edge of the precedence graph is directed from a philosopher having higher precedence to a philosopher having lower precedence. A hungry philosopher having higher precedence than all its neighbors is allowed to eat. The algorithm manipulates the acyclicity of the precedence graph so that every hungry philospher eventually takes precedence over all its neighbors and eats.

The algorithm describes a way to implement the precedence graph in a distributed manner so that all changes to the edge directions can be made locally. To this end, all forks are assumed to be either *clean* or *dirty*. A clean fork remains clean until it is used for eating. The act of eating with a fork dirties it. A philosopher cleans a fork only when sending it to its neighbor. The direction of an edge between two neighboring philosophers u and v is directed from u to v if and only if

- u holds the common fork and the fork is clean or,
- v holds the common fork and the fork is dirty.

Note that only eating can change the orientation of the edges of the precedence graph. Since eating dirties the forks used, an eating philosopher has *all* its incident edges directed towards it. Redirecting *all* the edges of a vertex of a graph to point towards it cannot create any new cycles involving that vertex. So if the precedence graph was acyclic before the philosopher ate, it continues to remain so. Initially all forks are dirty and are located at philosophers such that the precedence graph is acyclic. Hence acyclicity is an invariant.

A hungry philosopher requisitions the forks he needs by sending out request tokens to the philosophers that hold the forks. A philosopher holding both the fork and the corresponding request token, releases the fork only if the requisitioning philosopher has higher precedence than it. It can be shown that every hungry philosopher receives the forks it has requested and eventually eats.

The starting state of the algorithm is as follows.

- All philosophers are thinking.
- For each edge, the fork and request token are held by different philosophers.
- All forks are dirty.
- The forks are placed in such a manner that the precedence graph is acyclic.

The algorithm is described in the appendix. For the purposes of our development, we note that the algorithm has the following properties.

- *Mutual Exclusion* : Neighboring philosophers do not eat simultaneously.
- *Starvation Freedom* : Every hungry philosopher eats eventually.

The interested reader is referred to [CM84, CM88] for an excellent exposition and proof of this algorithm.

5 The Eventually-Determinizing Algorithm

Now we show how to merge the algorithms of Section 3 and 4 to obtain an eventually-determinizing algorithm for the dining philosophers problem. It is required that the starting state of the algorithm be identical for all philosophers. Except for the positioning of the dirty forks, there is no conflict in the starting states of the probabilistic and deterministic algorithms. Thus they can be consistently merged.

- All philosophers are thinking.
- No philosopher has made a random choice.

- For each edge, the fork and the request token are with different philosophers. In particular, a philosopher holds the fork it shares with its right neighbor and the request token it shares with its left neighbor.
- All forks are dirty.
- All request tokens are clean.
- The $rffirst$ attribute of the request token is true.

During the execution of an eventually-determinizing algorithm, it is possible to have a mix of probabilistic and deterministic philosphers. Since a philosopher executing in a particular mode expects certain responses from its environment it is necessary that the interface between the modes be carefully designed. For example, a philosopher executing in a probabilistic mode should not have to wait, if the second fork is "in use". Similiarly, a philosopher in the deterministic mode should not have the dirty request token returned to it. It waits for all the forks that it requests.

A close inspection of the variables defined and their occurrence in the algorithms helps us solve the problem. Notice that the attributes of the request token ($rfclean[u, v]$ and $rffirst[u, v]$) are only of relevance to the probabilistic algorithm while the attribute of the fork ($clean[u, v]$) is only used in the deterministic algorithm. We propose the following two rules.

- Before sending a request token to philosopher v, a deterministic philosopher u sets the $rffirst[u, v]$ attribute to true.
- A probabilistic philosopher u dirties a fork when sending it to philospher v.

The first rule ensures that if v is a probabilistic philospher, it will interpret the *true* value $rffirst[u, v]$ attribute to mean that u will wait for the fork. Thus the v does not send a dirty request token to u. The case where the v is a deterministic philosopher is easily ruled out as the $rffirst$ attribute has no relevance for such a philosopher.

The second rule is more subtle. It is designed to ensure that if a probabilistic philosopher ever has higher precedence than a deterministic philosopher then it will continue to do so until it becomes deterministic. That is, by the rules of precedence of the Chandy-Misra algorithm, philosopher u will have higher precedence than v. Suppose v is a deterministic philosopher. From the rules of the deterministic algorithm, if u ever requests the fork from v, it will get it back immediately. This is particularly useful when $(draw.u = R(v, u))$. In such a case, u will not have to wait for its second fork. The problem of a deterministic philosopher having to send a dirty request token to a probabilistic philosopher has vanished ! The case where v is a probabilistic philosopher is easily ruled out as the $clean$ attribute has no relevance for such a philosopher.

The above two rules allow a uniform interface to be constructed between two philosophers executing in different modes. We now present the conditions under which a philosopher can switch its mode from probabilistic to deterministic. The rules are

- A probabilistic philosopher who eats, dirties his forks and becomes deterministic.
- Upon receipt of a clean fork, a probabilistic philospher becomes deterministic.

We informally argue about the soundness of these rules. Given the starting state of the algorithm, by the rules of the Chandy-Misra algorithm, the precedence graph is a cycle. If a philosopher becomes hungry, by the Lehman-Rabin algorithm some other philosopher eats. By the first rule, he dirties his forks and becomes deterministic. Since both the incident edges now point towards it, the precedence graph is acyclic. At this point, there exists a deterministic philosopher whose precedence is lower than that of its neighbors. We claim that this condition is stable.

The only way that the direction of an edge can be changed is if a philosopher eats or if a probabilistic philosopher sends a dirty fork to its neighbor. In the first case, the philosopher who eats, dirties his forks and becomes deterministic. This only validates the claim further. In the second case, suppose that the probabilistic philosopher sends a dirty fork to a deterministic neighbor. The direction of an edge between a probabilistic and deterministic philosopher has been changed to point towards the deterministic philosopher. This does not invalidate the claim. The case of a probabilistic philosopher sending a dirty fork to another probabilistic philosopher can be ignored. Thus the precedence graph continues to be acyclic.

Note that the forks are initially dirty and can only be cleaned by deterministic philosophers. Thus the receipt of a clean fork assures a philosopher that determinism has been achieved in the ring. This alongwith the above stable condition guarantees that the precedence graph is acyclic. It is safe for the receiving philosopher to switch modes.

We sketch an argument to show that every hungry philospher eventually eats.

Lemma 0 *If a hungry probabilistic philosopher u chooses to wait for a $fork[u, v]$, then, with probability one, either u gets the fork or philosopher v eats.*

Lemma 1 *Let u be a probabilistic philosopher and let v be a neighboring deterministic philosopher. If the common fork is at v then the fork is dirty.*

Lemma 2 *If a hungry probabilistic philospher u has a deterministic neighbor, then either u eats or becomes hungry in the deterministic mode.*

Proof Sketch: Let t, u and v be three consecutive philosophers. Let u be the hungry, probabilistic philosopher and let v be its deterministic neighbor. If $draw.u = v$ and v possess $fork[u, v]$, then due to Lemma 1, u will succeed in getting a clean fork from v. If it fails in getting $fork[t, u]$, it will become hungry in the deterministic mode.

If $draw.u = v$ and u possesses $fork[u, v]$ then with probability one, philosopher u's random choice will become t. Thus philosopher u will wait for $fork[t, u]$ and by Lemma 0 will either get the $fork[t, u]$ or t will eat. In the latter case, t becomes determinisitic and by the rules of the deterministic algorithm, t will send fork $fork[t, u]$ to u. Recall, that the a probabilistic philosopher dirties his forks when sending them to his neighbor. Thus a probabilistic philosopher u always has precedence over deterministic philosopher v, and will succeed in getting $fork[u, v]$ and eating. (End of Proof Sketch)

Lemma 3 *Let u, v and w be three consecutive probabilistic philosophers. Let v be hungry and w be thinking. Then, with probability one, either one of u or v will eat or w will become hungry.*

Lemma 4 *Every philosopher who is hungry in the probabilistic mode, either eats or becomes hungry in the deterministic mode.*

Proof Sketch: Let u, v and w be three consecutive philosophers. Let v be the hungry probabilistic philosopher. By the Lehman-Rabin proof, some philosopher z eats and becomes deterministic. Assume that none of v's neighbors is deterministic and one neighbor (say, w) is thinking. Then Lemma 3, either u eats or v eats or w becomes hungry in the probabilistic mode. By a simple induction argument, a chain of hungry probabilistic philosophers is set up until a deterministic philosopher is encountered. By Lemma 2, either the last hungry probabilistic philosopher on the chain (who has z for a neighbor) eats or becomes hungry in the deterministic mode. In either case, it becomes deterministic and the contagion of determinism flows back along the chain. Eventually v eats or becomes hungry in the deterministic mode. (End of Proof Sketch)

Lemma 5 *Every philosopher who is hungry in the deterministic mode eats.*

Proof Sketch: Let v be a hungry, deterministic philosopher. If there are no probabilistic philosophers in the ring, since the precedence graph is acyclic, the starvation-freedom proof of Chandy-Misra is applicable and eventually v eats. Otherwise, let M be the number of probabilistic philosophers in the ring, N the number of deterministic ancestors of u in the precedence graph and L the number of thinking deterministic ancestors of u in the precedence graph. By an induction argument, we show that the metric (M, N, L) decreases until u eats. (End of Proof Sketch)

Theorem 0 *No philosopher is hungry forever.*

The eventually-determinizing algorithm appears in the appendix. The author has proved it in a compositional manner by stating the proofs of the Lehman-Rabin and the Chandy-Misra algorithms as conditional properties and using a calculus that he has developed for manipulating such properties.

6 Conclusions and Future Research

We have introduced a new paradigm for the design of parallel algorithms and applied it to design an eventually-determinizing algorithm for a circular configuration of dining philosophers. Such algorithms utilise the advantages of both probabilistic and deterministic algorithms.

The paradigm opens up several interesting problems in the theory and practice of algorithm design. A first question tries to generalize the paradigm further. Can the idea of mode switching be applied to other properties, like asynchrony and synchrony ? Secondly, a theory of eventually-determinizing algorithms would be useful. It would help in answering questions such as: What is the weakest specification that a probabilistic mode has to satisfy for its combination with a given deterministic mode to be eventually determinizing ? In our example, we have assumed the probabilistic mode to satisfy deadlock-freedom. Would something weaker have done the job ? Another question relates to other applications of eventual determinism. Currently the author is working on applying the paradigm to mutual exclusion and network routing.

A related area of study would be to define suitable complexity measures for such algorithms. It is evident that something more than the expected value analysis of probabilistic algorithms and the lower and upper bound measures of deterministic algorithms is needed.

7 Acknowledgement

I am grateful to Professor Jayadev Misra for advising and supporting me. Thanks are also due to Ken Calvert, Sumit Ganguly, K. Muthukumar and Sankrant Sanu for their suggestions and criticisms.

References

[BGS88] Shaji Bhaskar, Rajive Gupta, and Scott Smolka. Probabilistic algorithms: A survey. Private Communication, 1988.

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6:632–646, October 1984.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Dij72] Edsger W. Dijkstra. *Operating Systems Techniques*, chapter Hierarchical Ordering of Sequential Processes. Academic Press, New York, 1972.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, November 1974.

[DS89] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[HSP83] Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5:356–380, 1983.

[Lev88] Eliezer Levy. A probabilistic-deterministic algorithm, or an exercise in program composition. Private Communication, 1988.

[LR81] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 133–138, Williamsburg, VA, 1981.

[Pnu83] Amir Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the 15th Annual Symposium on the Theory of Computing*, pages 278–290, 1983.

[PZ86] Amir Pnueli and Lenore Zuck. Verification of multiprocess protocols. *Distributed Computing*, 1:53–72, 1986.

[Rab76] Michael O. Rabin. *Algorithms and Complexity*, chapter Probabilistic Algorithms, pages 21–40. Academic Press, New York, 1976.

[Rao90] Josyula R. Rao. Reasoning about probabilistic algorithms. Submitted to the 9th Annual ACM Symposium on Principles of Distributed Computing, 1990.

8 Appendix

8.1 The Lehman-Rabin Algorithm

The Lehman-Rabin algorithm is formalized in a UNITY-based notation designed by the author [Rao90]. The interested reader is referred to [LR81, PZ86] for an alternate exposition.

always

$\langle \; [] \; u ::$

$$
\begin{array}{lll}
u.mayeat & = & u.h \wedge (draw.u \neq N) \wedge (fork[u, u-1] = u) \wedge (fork[u, u+1] = u) \\
[] \quad u.retry & = & u.h \wedge (draw.u \neq N) \wedge (fork[u, draw.u] = u) \wedge \\
 & & (fork[u, R(draw.u, u)] \neq u) \wedge (rf[u, R(draw.u, u)] = u) \wedge \\
 & & \neg rfclean[u, R(draw.u, u)]
\end{array}
$$

$\rangle$

$[] \quad \langle \; [] \; (u, v) : E[u, v] :$

$$
\begin{array}{lll}
sendreq[u, v] & = & u.h \wedge (draw.u \neq N) \wedge (fork[u, v] = v) \wedge (rf[u, v] = u) \wedge rfclean[u, v] \\
[] \quad sendfork[u, v] & = & (fork[u, v] = u) \wedge (rf[u, v] = u) \wedge \\
 & & (u.t \vee (u.h \wedge (draw.u = N)) \vee (u.h \wedge draw.u = R(v, u) \wedge fork[u, draw.u] \neq u \\
[] \quad forkinuse[u, v] & = & (fork[u, v] = u) \wedge (rf[u, v] = u) \wedge \\
 & & (u.e \vee (u.h \wedge (draw.u = v)) \vee (u.h \wedge draw.u = R(v, u) \wedge fork[u, draw.u] = u
\end{array}
$$

$\rangle$

initially

$\langle \; [] \; u ::$

$u.dine, draw.u \; = \; t, N$

$[] \quad fork[u, u+1] \; = \; u$

$[] \quad rf[u, u+1], rfclean[u, u+1], rffirst[u, u+1] \; = \; (u+1), true, true$

$\rangle$

assign

$draw.u \; := \; u - 1 \mid u + 1 \quad \textbf{if} \quad u.h \wedge (draw.u = N)$

$[]$

$rf[u, draw.u], rffirst[u, draw.u] \; := \; draw.u, true \quad \textbf{if} \; sendreq[u, draw.u]$

$[]$

$rf[u, R(draw.u, u)], rffirst[u, R(draw.u, u)] \; := \; R(draw.u, u), false \quad \textbf{if} \; sendreq[u, R(draw.u, u)] \; (fork[u, draw.u] = u)$

$[]$

$u.dine := e \quad \textbf{if} \quad u.mayeat$

$[]$

$draw.u, rfclean[u, R(draw.u, u)] := N, true \quad \textbf{if} \quad u.retry$

$[]$

$\langle \; [] \; v : E[u, v] :$

$fork[u, v], rf[u, v], rfclean[u, v]$

$:= fork[u, v], v, false \quad \textbf{if} \quad forkinuse[u, v] \wedge \neg rffirst[u, v] \sim$

$v, rf[u, v], rfclean[u, v] \quad \textbf{if} \quad sendfork[u, v]$

$\rangle$

end

8.2 The Chandy-Misra Algorithm

We describe the deterministic Chandy-Misra algorithm in UNITY. The interested reader is referred to [CM84, CM88] for an excellent exposition of this work.

always

$$
\begin{array}{lll}
\langle \,[]\; u :: & u.mayeat & = \langle \wedge v : E[u,v] : (fork[u,v] = u) \wedge (clean[u,v] \vee (rf[u,v] = v)) \rangle \\
\rangle & & \\
[] \quad \langle \,[]\; u,v : E[u,v] :: & & \\
\quad sendreq[u,v] & = & (fork[u,v] = v) \wedge (rf[u,v] = u) \wedge u.h \\
[] \quad sendfork[u,v] & = & (fork[u,v] = u) \wedge \neg clean[u,v] \wedge (rf[u,v] = u) \wedge \neg u.e \\
\rangle & &
\end{array}
$$

initially

$$
\begin{array}{l}
\langle \,[]\; u :: u.dine = t \rangle \\
[] \quad \langle \,[]\; (u,v) :: clean[u,v] = false \rangle \\
[] \quad \langle \,[]\; (u,v) : u < v :: fork[u,v], rf[u,v] = u, v \rangle
\end{array}
$$

assign

$$
\begin{array}{lll}
\langle \,[]\; u :: & u.dine := e & \textbf{if } u.h \wedge u.mayeat \\
& \|\langle \| v : E[u,v] :: clean[u,v] := false & \textbf{if } u.h \wedge u.mayeat \rangle \\
\rangle & & \\
[] \quad \langle \,[]\; (u,v) :: & & \\
& rf[u,v] := v & \textbf{if } sendreq[u,v] \\
\quad [] & fork[u,v], clean[u,v] := v, true & \textbf{if } sendfork[u,v] \\
\rangle & &
\end{array}
$$

end

8.3 The Eventually-Determinizing Algorithm

The eventually-determinizing algorithm is shown in its entirety below. It is obtained by merging the UNITY-like notations of the modified Lehman-Rabin and the Chandy-Misra algorithms and incorporating the mode switching rules developed in Section 5.

always

$$
\begin{array}{lll}
\langle \,[]\; u :: & & \\
\quad u.mayeatp & = & u.h \wedge u.p \wedge (draw.u \neq N) \wedge (fork[u, u-1] = u) \wedge (fork[u, u+1] = u) \\
[] \quad u.retry & = & u.h \wedge u.p \wedge (draw.u \neq N) \wedge (fork[u, draw.u] = u) \wedge \\
& & (fork[u, R(draw.u, u)] \neq u) \wedge (rf[u, R(draw.u, u)] = u) \wedge \\
& & \neg rfclean[u, R(draw.u, u)] \\
\quad u.mayswitch & = & \langle \vee v : E[u,v] : (fork[u,v] = u) \wedge clean[u,v] \rangle \\
\rangle & & \\
[] \quad \langle \,[]\; (u,v) : E[u,v] : & & \\
\quad psendreq[u,v] & = & u.h \wedge u.p \wedge (draw.u \neq N) \wedge (fork[u,v] = v) \wedge (rf[u,v] = u) \wedge rfclean[u, \\
[] \quad psendfork[u,v] & = & (fork[u,v] = u) \wedge (rf[u,v] = u) \wedge u.p \wedge \\
& & (u.t \vee (u.h \wedge (draw.u = N)) \vee (u.h \wedge draw.u = R(v,u) \wedge fork[u, draw.u] \neq u \\
[] \quad forkinuse[u,v] & = & (fork[u,v] = u) \wedge (rf[u,v] = u) \wedge u.p \wedge \\
& & (u.e \vee (u.h \wedge (draw.u = v)) \vee (u.h \wedge draw.u = R(v,u) \wedge fork[u, draw.u] = u) \\
\rangle & &
\end{array}
$$

$\langle \,[]\; u :: \quad u.mayeatd \quad = \quad u.d \wedge \langle \wedge v : E[u,v] : (fork[u,v] = u) \wedge (clean[u,v] \vee (rf[u,v] = v)) \rangle$

$\rangle$

$[] \quad \langle \,[]\; u, v : E[u,v] ::$

$\quad dsendreq[u,v] \quad = \quad (fork[u,v] = v) \wedge (rf[u,v] = u) \wedge u.h \wedge u.d$

$[] \quad dsendfork[u,v] \quad = \quad (fork[u,v] = u) \wedge \neg clean[u,v] \wedge (rf[u,v] = u) \wedge \neg u.e \wedge u.d$

$\rangle$

initially

$\langle \,[]\; u ::$

$\quad u.dine, u.mode, draw.u \;=\; t, p, N$

$[] \quad fork[u, u+1], clean[u, u+1] \;=\; u, false$

$[] \quad rf[u, u+1], rfclean[u, u+1], rffirst[u, u+1] \;=\; u+1, true, true$

$\rangle$

assign

$draw.u \;:=\; u-1 \mid u+1 \quad \textbf{if} \quad u.h \wedge u.p \wedge (draw.u = N)$

$[]$

$rf[u, draw.u], rffirst[u, draw.u] \;:=\; draw.u, true \quad \textbf{if}\; psendreq[u, draw.u]$

$[]$

$rf[u, R(draw.u, u)], rffirst[u, R(draw.u, u)] \;:=\; R(draw.u, u), false \quad \textbf{if}\; psendreq[u, R(draw.u, u)] \wedge (fork[u, draw.u] = u)$

$[]$

$u.dine, u.mode, clean[u, u-1], clean[u, u+1] :=$

$\quad e, p, false, false \quad \textbf{if} \quad u.mayeatp$

$[]$

$draw.u, rfclean[u, R(draw.u, u)], u.mode :=$

$\quad N, true, d \quad \textbf{if} \quad u.retry \wedge u.mayswitch \sim$

$\quad N, true, u.mode \quad \textbf{if} \quad u.retry \wedge \neg u.mayswitch$

$[]$

$\langle \,[]\; v : E[u,v] :$

$\quad fork[u,v], rf[u,v], rfclean[u,v], clean[u,v]$

$\quad := fork[u,v], v, false, clean[u,v] \quad \textbf{if} \quad forkinuse[u,v] \wedge \neg rffirst[u,v] \sim$

$\quad v, rf[u,v], rfclean[u,v], false \quad \textbf{if} \quad psendfork[u,v]$

$\rangle$

$[] \quad \langle \,[]\; u :: \quad u.dine := e \quad \textbf{if} \quad u.h \wedge u.mayeatd$

$\quad \| \langle \| v : E[u,v] :: clean[u,v] := false \quad \textbf{if} \quad u.h \wedge u.mayeatd \rangle$

$\rangle$

$[] \quad \langle \,[]\; (u,v) ::$

$\quad rf[u,v], rffirst[u,v] := v, true \quad \textbf{if} \quad dsendreq[u,v]$

$[] \quad fork[u,v], clean[u,v] := v, true \quad \textbf{if} \quad dsendfork[u,v]$

$\rangle$

end