Parallel Processing Letters Vol. 8, No. 4 (1998) 421–432

SPECIFICATION AND VERIFICATION OF THE ATMR PROTOCOL USING UNITY

MICHEL CHARPENTIER
and
GÉRARD PADIOU
IRIT/INPT
2, rue Charles Camichel, BP 7122
31071 Toulouse cedex 7, France

Received September 1997
Revised March 1998
Accepted by M. Cosnard

Part 1: modeling and specification of the protocol

ABSTRACT

We propose a description and validation of the ATMR protocol within the UNITY formalism. This formal description helps us understand precisely the mechanisms this protocol involves. In particular, we have noted the use of an incorrect detection algorithm to generate reset slots. In this first part, we provide an operational description using the UNITY programming notation as well as a specification of the main correctness properties in the UNITY temporal logic. A second part [2] is dedicated to a hand-made correctness proof. The proof shows that the model satisfies the specification despite this spurious detection. In this study, we apply a general development process based upon the UNITY formalism. During this process, we tune the program specification in order to make the later validation step easier, in the same way as the inclusion of traces, breakpoints, and assertions prepares a program for its quality assurance tests.

Keywords: UNITY, protocol validation, ATM protocols, temporal logic, formal methods

1 Introduction

Communication protocols are examples of concurrent and reactive systems. The proof of their correctness is a difficult but indispensable task. While a great deal of research has been done in this domain, with normalized and industrial protocols, simulation by means of specialized tools is still the most widely used method of validation.

Our proposal investigates a formal approach based upon the Unity model. We apply this formalism to an actual protocol, namely the ATMR (Asynchronous Transfer Mode Ring) protocol, to illustrate a general approach to reactive and concurrent program verification.

After a short informal description of the ATMR protocol and a brief presentation of the Unity formalism, we follow a general validation process. This development involves the following steps:

- First, we use the Unity programming notation to describe an operational model of the protocol. This model is implemented as a parallel program. At this level, the consistency of the program is validated with respect to the Unity semantics.
- Next, the correctness proof requires a preliminary "decoration" of the Unity program to develop the specifications and their proofs. This crucial tuning step consists in *superposing* auxiliary variables which capture attributes (properties) used within the proofs.
- We then specify the correctness requirements of the protocol (and thus, of the program) using the Unity logic.
- Then, we prove a set of elementary safety and liveness properties with respect to the program. While these simple proofs are cumbersome, proof assistants are available to help.
- Lastly, the correctness proofs of main specifications are done by hand.

This paper develops the first three points. The two remaining points are dealt with in the second part [2].

2 The ATMR Protocol

2.1 Basic Principles of High Speed Local-Area Networks

The technological evolution in multimedia applications has introduced new requirements for communication resulting in the development of very high bandwidth networks. Today, ATM (Asynchronous Transfer Mode)-based transfer of information is the primary technique employed in wide-area networks and is also used in high-speed local-area networks. Information is carried in small packets of fixed length, called cells. While nodes send and receive cells asynchronously, the transmission stream must be a continuous circuit. When there is no data ready to be sent, empty cells are transmitted. This means that in the physical layer, ATM is synchronous [7].

In the local domain, where a new generation of LANs has been developed to extend bandwidth to the gigabit range, the ring is a typical topology. It offers several advantages in reliability, ease of optical fiber use, flexibility in the length of the physical medium, and spatial reuse of slots by destination release. In the first generation of ring-based networks, the information sent by a node is deleted by the sending node only after a complete rotation around the ring. With spatial reuse, the destination node removes the information. Because concurrent accesses with spatial reuse are allowed, throughput can be improved.

2.2 Overview of the ATMR Media Access Mechanism

For controlling access to this type of shared medium, several medium access control protocols have been designed. Within this framework, ATMR is a new ISO standard [5].

The protocol's design is based upon a slotted-ring with spatial reuse [4]. The ring is first initialized to contain a fixed number of bits grouped into fixed length slots. These slots continuously circulate around the ring. A slot is exactly the ATM cell size (5-byte header, 48-byte information field). Compared to other similar protocols, ATMRś design adopts a distributed scheme: the nodes do not have to exchange information with a specific node or with each other to transmit a message. The protocol is based on a window mechanism and a distributed reset procedure.

2.3 Transmission and Reception Mechanisms

A node transmits a cell in an empty slot and adds the destination address and its own address in specific fields of the slot. The names of these fields are the destination address (DA) and the busy address (BA), respectively. When a non-empty slot arrives, a node looks for its own address in the destination field of the slot. If there is a match, the information is received and the destination address is reset to zero so that the slot becomes empty again.

2.4 Window Mechanism

When a visiting slot is empty, it is available for use by a node that has data to transmit and a positive credit. Initially, each node is given an amount of credit, called its *window size*. A node's window size indicates the number of cells it is allowed to transmit before it must be reset. It is recorded in a local window counter, which is decremented by one for every transmitted cell. When a node's credit reaches zero, the node stops sending cells, even if there are still cells waiting to be transmitted. Upon the arrival of a special cell, named *reset*, the window counter is reset to its initial value and the transmission of cells can be restarted.

2.5 Reset Mechanism

When a node is active, that is to say there exist pending messages and the window counter is positive, it overwrites the busy address field (BA) of all visiting slots with its own address.

A reset cell is issued from a node that finds its own address in the busy address field of the incoming slot. It deduces the (possibly wrong, see [2]) transmission completion of all other nodes, and generates a reset cell. Any node generating such a cell has to remove it from the ring. Therefore, the node enters a so-called *hunting state* until the removal of the first visiting reset cell.

3 The UNITY Formalism

The purpose of the UNITY formalism [1] is to provide a framework for reasoning about the *correctness* of programs. It is a general model that supports the specification of parallel programs from an operational point of view (writing statements) as well as from a declarative point of view (stating properties). The formalism is designed to check the consistency between these two points of view.

3.1 UNITY Programs

The UNITY programming language is based on fair transition systems. A program consists of four sections:

- the **declare** section defines the set of state variables along with their types;
- the **initially** section consists of a predicate on state variables expressing the acceptable initial states for the program;
- the **always** section defines some variables as functions of others (macro functionality);
- the **assign** section contains the program statements. More precisely, if contains a set of multiple assignment statements. The symbol $[\!]$ is used to separate these assignments.

A program execution runs forever, starting from any state satisfying the initial condition. At each step, a statement is arbitrarily selected for execution. Moreover, each statement is executed infinitely often (weak fairness hypothesis). Terminating programs are interpreted as programs that reach a fixed point (a stable state). On reaching such a state, any further execution step leaves the state unchanged.

The only UNITY statement is the assignment statement. Assignment has several forms: conditional, quantified, and enumerated. For instance, a conditional multiple assignment has the following syntax:

$$\begin{array}{lclll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & \texttt{if} & g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & \texttt{if} & g_2 \\ & \cdots & \cdots & \texttt{if} & \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & \texttt{if} & g_p \end{array}$$

Unlike Dijkstra's guarded commands, when several conditions are simultaneously true, the assigned values must be the same. The assignment leaves the variable values unchanged if none of the conditions evaluates to true. Therefore, each assignment is deterministic. Of course, due to the arbitrary selection of statements, UNITY programs are, in general, non-deterministic.

Enumerated assignments are a useful syntactic feature to shorten long multiple assignments. For instance, $x := a \parallel y, z := b, c \parallel t := d$ is an equivalent notation for $x, y, z, t := a, b, c, d$.

Quantification is allowed, especially with parallel or exclusive assignments. For instance, the quantified assignment $\langle \parallel i : 1 \leqslant i \leqslant N :: A[i] := B[i] \rangle$ is equivalent to $A[1] := B[1] \| A[2] := B[2] \| \cdots \| A[N] := B[N]$.

In this paper, we consider UNITY programs from an operational point of view [8]. The semantics $\mathcal{O}.F$ of a program F is the set of all its possible computations. A computation σ is a sequence of states σ_i. Each state derives from its predecessor by executing a statement in F. For an expression e, the notation $\sigma_i.e$ denotes the value of the expression e in the state σ_i.

3.2 The UNITY Logic

A specification language is associated with the former operational language. It allows safety and liveness properties to be specified as temporal logic expressions. The program variables are operands of such predicates.

The definition given here for these properties is purely operational and differs from the original definition of [1]. From the operational point of view, UNITY relations are properties of the semantics of a program. That is to say, they are properties of the set of all possible computations. We find it useful when building proofs in [2] to reason on a single computation and to express properties of such a computation in the same way we would have expressed it for the set of all computations. That is why we choose to introduce UNITY-like properties on a single computation.

We denote the relations defined on a computation and the relations defined on the set of computations with the same keyword. Relations defined on computations are written in lower case, while relations defined on the set of computations are written in upper case. Of course, the theorems of the UNITY logic apply to the relations defined on the set of computations, which are equivalent to Sanders' properties [6]. On the other hand, the formalization of the UNITY logic with the prover NQTHM [3] shows that the theorems are also satisfied by the relations over a single computation. These relations over a single computation enjoy useful additional properties.

Safety property syntax:

Three basic relations allow to express safety properties, namely: *unless*, *stable* and *invariant*. For a computation σ and predicates P and Q, they are defined as follows:

- p **unless** q: if p becomes true, it remains true at least until q becomes true:

$$p \textbf{ unless } q \;\; \texttt{in} \;\; \sigma \;\equiv\; \langle \forall i :: \;\; \sigma_i.(p \wedge \neg q) \Rightarrow \sigma_{i+1}.(p \vee q)\rangle$$

- **stable** p: once p becomes true, it remains true:[a]

$$\begin{aligned}\textbf{stable } p \;\; \texttt{in} \;\; \sigma &\equiv \langle \forall i :: \;\; \sigma_i.p \Rightarrow \sigma_{i+1}.p\rangle \\ &\equiv \langle \forall i :: \;\; \sigma_i.p \Rightarrow \langle \forall j : j \geqslant i :: \;\; \sigma_j.p\rangle\rangle\end{aligned}$$

- **invariant** p: p is true initially and it is stable. Therefore, p is always true:

$$\begin{aligned}\textbf{invariant } p \;\; \texttt{in} \;\; \sigma &\equiv \sigma_0.p \;\wedge\; \textbf{stable } p \;\; \texttt{in} \;\; \sigma \\ &\equiv \langle \forall i :: \sigma_i.p\rangle\end{aligned}$$

Liveness property syntax:

The fundamental liveness property of the UNITY logic is the *leads-to* property. A *leads-to* property expresses that a predicate eventually becomes true after a finite time. We also use a stronger property called *until* which contains both a safety part and a liveness part. Their definitions are:

- p **leads-to** q: if p is true, then q will eventually become true:

$$p \textbf{ leads-to } q \;\; \texttt{in} \;\; \sigma \;\equiv\; \langle \forall i :: \;\; \sigma_i.p \Rightarrow \langle \exists j : j \geqslant i :: \;\; \sigma_j.q\rangle\rangle$$

- p **until** q: if p is true, then it remains true until q becomes true, and eventually, q becomes true:

$$\begin{aligned}p \textbf{ until } q \;\; \texttt{in} \;\; \sigma &\equiv p \textbf{ unless } q \;\; \texttt{in} \;\; \sigma \;\wedge\; p \textbf{ leads-to } q \;\; \texttt{in} \;\; \sigma \\ &\equiv \langle \forall i :: \sigma_i.p \Rightarrow \langle \exists j : j \geqslant i :: \sigma_j.q \wedge \langle \forall k : i \leqslant k < j :: \sigma_k.p\rangle\rangle\rangle\end{aligned}$$

A relation on the set of computations is defined from the corresponding relation on a single computation in the same way for all relations. For instance:

$$p \text{ UNLESS } q \;\; \texttt{in} \;\; F \;\equiv\; \langle \forall \sigma : \sigma \in \mathcal{O}.F :: p \textbf{ unless } q \;\; \texttt{in} \;\; \sigma\rangle$$

4 Modeling the ATMR Protocol

4.1 ATMR as a UNITY Program

ATMR is composed of two kind of entities: slots circulating on a ring network and nodes visited by the slots. The ring of slots is implemented as an array of M slots. We assume this number of slots is greater than the number of nodes. The nodes are implemented as an array of N nodes. Both of these arrays are global variables.

[a] **stable** p is equivalent to p **unless** $false$.

The statements define the behavior of the global system. This involves the behavior of nodes and the rotation of slots along the ring. The parallel composition of UNITY allows to describe the synchronous actions of the protocol. The dynamic behavior of the system is a sequence of slot visits. When a node is visited by a slot, the following steps occur:

- the node extracts a cell from the slot if the slot is full and the node is the right destination;
- the node deposits a cell into the visiting slot if it is empty and the node has both pending messages and a positive window counter;
- the slot leaves the node for the next node.

This approach provides a symmetric abstraction of the system: we can either consider that the slots continuously visit the nodes or that the nodes continuously move from slot to slot. Of course, the first interpretation is the actual system.

The UNITY program involves the following components:

- The declaration of global variables implementing the slots and the nodes. An extra variable $State$ is used to define the current state of the system. Three states are distinguished: $receiving$, $sending$, and $rotating$. During the $receiving$ and $sending$ states, each node handles its visiting slot. During the $rotating$ state, a one-slot global shift occurs.

  Each node contains an input and an output message queue. These queues are implemented with variables of type *sequence*. We use the classical operators *append*, *tail* (*tl*), *head* (*hd*), *length*, and *member*[b].

- An **initially** section specifying the starting global state of the system. Initially, all slots are empty, each node points to a different slot, and all nodes have window counters equal to $WSIZE > 0$.

- An **assign** section describing the global behavior of the system as a three-state automaton. A cyclical sequence of transitions is performed from $receiving$ to $sending$ to $rotating$.

We introduce a **with** syntax to avoid the explicit prefixing of the assignments with the array of nodes since the three statements of our program are identical for all nodes. When we need to handle different nodes and/or different slots in the same formula (as is the case in the specification and proofs), we avoid using the **with** syntax and we explicitly identify all arrays and indices.

The type of message content is irrelevant and is denoted by $Tdata$. In the same way, "_" represent the undefined initial content of messages in slots.

[b]The *append* operator is used to append a value at the end of a queue and not to concatenate two lists. In the same way, $member(x, l)$ is denoted by $x \in l$ to mean that x is an element of list l.

```
Program ATMR
Declare
  type Tslot is record
    adr     : 0..N;            -- slot destination address (DA)
    busy    : 0..N;            -- used in slots reservation mechanism (BA)
    reset   : bool;            -- used to reset window counters
    msg     : Tdata;           -- transmitted data
  end record;

  type Tnode is record
    sl      : 1..M;            -- index of the slot visiting the node
    wc      : int;             -- current window counter
    hunting : bool;            -- reset cell removal state
    out     : seq of Tdata;    -- output queue
    in      : seq of Tletter;  -- input queue
  end record;

  type Tletter is record
    adr     : 1..N;            -- destination address
    data    : Tdata;           -- message content
  end record;

  type Tstate is (receiving, sending, rotating);

    Slots   : array [1..M] of Tslot;
    Nodes   : array [1..N] of Tnode;
    State   : Tstate;
Initially
  State = receiving                                              ∧
  ⟨∀i : 1 ⩽ i ⩽ M :: Slots[i] = (0, 0, false, _)⟩                ∧
  ⟨∀s, s' : 1 ⩽ s, s' ⩽ N :: Nodes[s].sl = Nodes[s'].sl ⇒ s = s'⟩ ∧
  ⟨∀s : 1 ⩽ s ⩽ N :: with Nodes[s] do wc = WSIZE ∧ ¬hunting⟩;
Always
  ∀s : 1 ⩽ s ⩽ N :: withNodes[s] do
    Nbmsg   = length(in);
    Active  = wc > 0 ∧ Nbmsg > 0;
    Passive = ¬Active;
Assign
    ⟨⟨|| s : 1 ⩽ s ⩽ N :: with Nodes[s] do AtmrReceive(s)⟩ || State := sending⟩
                                  if State = receiving
  []
    ⟨⟨|| s : 1 ⩽ s ⩽ N :: with Nodes[s] do AtmrSend(s)⟩    || State := rotating⟩
                                  if State = sending
  []
    ⟨⟨|| s : 1 ⩽ s ⩽ N :: with Nodes[s] do AtmrRotate(s)⟩  || State := receiving⟩
                                  if State = rotating
End ATMR
```

This description is very concise because UNITY formalism allows us to clearly and simply describe the synchronous behavior of nodes. In a synchronous way, all nodes cyclically perform a *receiving* step, a *sending* step, and a *rotating* step. The receiving step handles the reception of a message and the reset slot generation or deletion. The sending step handles the sending of a message and the stamping of the busy address field when the node is active. Finally, the rotating step shifts all the slots in a circular fashion.

Statement $AtmrReceive(myself: 1..N)$ **is**
-- message reception
$Slots[sl].adr, out := 0, append(out, Slots[sl].msg)$
`if` $Slots[sl].adr = myself$
-- reception of a reset slot in non-hunting mode
$\|$ $wc := WSIZE$
`if` $Slots[sl].reset \wedge \neg hunting$
-- reception of a reset slot in hunting mode
$\|$ $Slots[sl].reset, hunting := false, false$
`if` $Slots[sl].reset \wedge hunting$
-- reset slot creation
$\|$ $Slots[sl].reset, wc, hunting := true, WSIZE, true$
`if` $Slots[sl].busy = myself \wedge Passive \wedge \neg hunting \wedge \neg Slots[sl].reset$[c]

Statement $AtmrSend(myself: 1..N)$ **is**
-- message sending
$Slots[sl].adr, Slots[sl].msg, wc, in := hd(in).adr, hd(in).data, wc - 1, tl(in)$
`if` $Slots[sl].adr = 0 \wedge Active$
-- slot stamping
$\|$ $Slots[sl].busy := myself$
`if` $Active \vee Hunting$

Statement $AtmrRotate$ **is**
-- slots shifting
$sl := (sl \bmod M) + 1$

4.2 Basic Consistency Validations

A set of basic consistency properties must be checked against this description. When a set of assignments are performed in a synchronous manner in UNITY, a single value must be assigned to each variable. This property is ensured because either the assigned variables are distinct or the guards of the assignments are mutually exclusive. In the statement *AtmrReceive*, statement parts assigning common references have exclusive guards. In the statements *AtmrSend* and *AtmrRotate*, differents parts never assign the same reference.

A further verification consists in checking that the nodes always handle different

[c]This last term of the conjunction can be omitted, but this can only be checked by proving a tricky invariant. Furthermore, in a real ATMR cell, the busy address field and the reset field are stored in the same place. This forces $busy = myself \Rightarrow \neg reset$.

slots. The following invariant formalizes this property with the *sl* field to point the current visiting slot.

$$\text{INVARIANT } \langle \forall s, s' \ : \ 1 \leqslant s, s' \leqslant N \ :: Nodes[s].sl = Nodes[s'].sl \Rightarrow s = s' \rangle$$

Initially, this property is true. After each synchronous handling of their visiting slot, the following "ticking step" is performed:

$$\langle \| \ s \ : \ 1 \leqslant s \leqslant N \ :: Nodes[s].sl := (Nodes[s].sl + 1) \bmod M \rangle$$

A *weakest precondition* evaluation allows the invariant to be proved.

4.3 Introducing Auxiliary Variables

A set of auxiliary variables are needed to describe and prove important properties. For instance, we would like to express that a message leaving a node reaches its destination after at most M slot shifts. In UNITY, a specific composition rule, the *superposition*, allows us to describe the introduction of auxiliary variables.

For the ATMR protocol, we first superpose a global clock variable. This clock ticks between each handling step synchronously performed by all nodes. This is achieved by adding to the statement *AtmrRotate* the following synchronous part:

$$\| \ Clock := Clock + 1$$

We also need to know the number of messages sent by each node. The record component *ec* (standing for *emission counter*) is incremented by one each time a node actually sends a message. The following part is added to the statement *AtmrSend*:

$$\| \ ec := ec + 1 \ \ \texttt{if} \ \ Slots[sl].adr = 0 \wedge Active$$

Finally, the number of passive nodes a slot visits consecutively is kept in the field *pc* (standing for *passive counter*). This counter is incremented by one when the visited node is passive and reset to zero if it is active. The following part is added to the statement *AtmrSend*:

$$\begin{array}{llll} \| \ \ Slots[sl].pc & := & Slots[sl].pc + 1 & \texttt{if} \ \ Passive \\ & \sim & 0 & \texttt{if} \ \ Active \end{array}$$

The properties stated in the proof are considerably simplified by using appropriate **always** definitions. Therefore, we introduce the three global counters WC, RC and HC that represent respectively the sum of the window counters of the different nodes, the number of reset slots on the ring, and the number of hunting nodes:

$$\begin{array}{l} \textbf{Always} \\ \ \ WC = \langle +s : 1 \leqslant s \leqslant N :: Nodes[s].wc \rangle \\ \ \ RC \ = \langle +j : 1 \leqslant j \leqslant M \wedge Slots[j].reset :: 1 \rangle \\ \ \ HC \ = \langle +s : 1 \leqslant s \leqslant N \wedge Nodes[s].hunting :: 1 \rangle \end{array}$$

5 Specifications

We now give the correctness specification for the protocol. This specification is a property of the program ATMR when it is composed by a union with its environment. Because message contents as well as destination addresses (provided they are between 1 and N) are irrelevant, we consider the following environment that appends undefined messages in the incoming queues in of the nodes:

$$\langle [\!] s : 1 \leqslant s \leqslant N :: Nodes[s].in := append(Nodes[s].in,\ (OneOf^{\mathrm{d}}\,(1..N), \_)) \rangle$$

As the main correctness property of any communication protocol, we require each sent message to be eventually received at the correct destination. With our notation, this property is expressed in the UNITY logic by the following relation:

$$\begin{array}{l} \forall s, s', m : 1 \leqslant s, s' \leqslant N :: \\ Nodes[s].Nbmsg > 0 \wedge hd(Nodes[s].in) = (s', m) \ \ \text{LEADS-TO} \ \ m \in Nodes[s'].out \end{array} \tag{1}$$

This property expresses that, if the node s wants to send the message m to the node s', then, eventually, the node s' will receive the message m.

6 Conclusions

This experiment illustrates the use of a formal method applied to a full-sized problem and the main features of the UNITY formalism. UNITY allows a concise specification of a relatively complex protocol.

The current formal description is the final result of several attempts. The main problem was the choice of the right granularity of synchronous actions. For instance, the handling of a visiting slot by a node could be described as a single atomic action. We discarded this solution because it implied a very intricate description of the reception and sending actions. The use of both the synchronous and asynchronous features of the UNITY formalism helped greatly in obtaining the right model.

We need to enrich the model with auxiliary variables to state and prove its correctness specification. Again, some UNITY features like the *superposition* rule are helpful. This step seems to us very important. It corresponds to the transformations applied to a program before it is tested, such as introducing breakpoints, traces, and so on, except that the transformation is used here to build a proof.

In the same way, the basic correctness property of the protocol has a very simple expression in the UNITY temporal logic. We now have to prove that the protocol description in terms of assignments satisfies this correctness property.

In a second part [2], we will present how the proof of this property can be performed. The proof process involves a combined bottom-up and top-down approach. On the one hand, some basic properties are extracted from the program text. On

[d] The underterministic *OneOf* function is used for readability. It could easily be implemented with deterministic UNITY statements.

the other hand, the correctness formula (1) is split into several lemmas to achieve the proof, with some lemmas being proved from the basic properties.

References

1. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
2. Michel Charpentier and Gérard Padiou. Specification and verification of the ATMR protocol using UNITY. Part 2: correctness proof. *Parallel Processing Letters*, 1999.
3. David M. Goldschlag. *Mechanically Verifying Concurrent Programs.* PhD thesis, University of Texas at Austin, May 1992.
4. K. Imai, T. Ito, H. Kasahara, and N. Morita. ATMR: Asynchronous transfer mode ring protocol. *Computer Networks and ISDN Systems*, 26:785–798, 1994.
5. ISO. Specification of the Asynchronous Transfer Mode Ring (ATMR) protocol (version 2.0), January 1993.
6. Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.
7. Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.
8. Rob T. Udink and Joost N. Kok. On the relation between UNITY properties and sequences of states. In J.W de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 594–608. Springer-Verlag, 1993.