Compositional BDD Construction: A Lazy Algorithm

J Bradley   N Davies

April 1998   **CSTR-98-005**

University of Bristol

Department of Computer Science

Also issued as ACRC-98:CS-005

Compositional BDD Construction: A Lazy Algorithm

Jeremy Bradley[†] and Neil Davies[‡]

Department of Computer Science,
University of Bristol, UK

6 April 1998

Abstract

The efficient Binary Decision Diagram algorithm presented in Brace, Rudell and Bryant 1990 [3] is modified to operate in a *lazy* or demand-driven fashion. We show how this new approach benefits the area of symbolic model checking by reducing the amount of state-space that needs to be evaluated. The need for a lazy or "delayed" BDD-based manipulation system, specifically to aid the reasoning about symbolic systems, was identified in Bryant 1992 [5] as a method of tackling the state-space explosion problem.

Often the model checking of systems only uses a very small part of the BDD (Bryant 1992 [5]). This is especially true for the generation of model checking counter-examples (Clarke et al 1994 [9]). However the strict nature of traditional BDD algorithms means that the whole BDD-representation of a system has to be evaluated.

Since this new technique only evaluates the nodes of the BDD that are required, it can potentially manipulate significantly larger and more complex systems than could be represented before.

We give space and time complexity comparisons between the traditional and lazy algorithms and show how it affects the practical evaluation of state-space especially for model checking purposes.

[†] Jeremy.Bradley@bristol.ac.uk
[‡] Neil.Davies@bristol.ac.uk

1 Introduction

The efficient representation and manipulation of Boolean expressions is a key consideration in the practicality of many computer-aided design tasks (Dill 1993 [10]).

In recent years, symbolic model checking (Clarke et al 1986 [8]) of sequential and concurrent systems has been an area in which researchers have used Boolean expressions to represent both systems and modelling properties (Burch et al 1990a [7], Burch et al 1990b [6]). However, these problems, like those relating to hardware verification, suffer from a state-space explosion when representing the actual systems (Bryant 1992 [5]).

Evaluating these systems and performing analysis on them is an inherently difficult task. Symbolic representations have exponential space complexity when expressed in DNF or sum-of-products form (Andersen 1996 [2]). Alternatively, if such a system is represented in CNF or product-of-sums form, the model checking problem can be NP-complete. An example of this is the satisfiability problem for CNF (Wilf 1986 [13]).

As a result Binary Decision Diagrams (BDDs) have long appealed as an alternative method of representing Boolean expressions. By exploiting uniformities in structure, they attempt to contain the space complexity of the symbolic representation of a system (Dill 1993 [10]). However BDDs too have worst-case exponential size (Akers 1978 [1]) and the main algorithms for constructing BDDs are strict in nature, since they require that the BDD is always fully evaluated. This is clearly unnecessary if the model checking or verification task only makes use of a small proportion of the BDD.

Each model checking test requires a necessary subset of the BDD to complete its task; our objective in this project was to develop an algorithm which just evaluates this subset and does so in a lazy fashion. In achieving this, we also potentially reduce the amount of system state-space that needs to be evaluated and so attempt to side-step the state-space explosion problem.

Our lazy technique forms part of a larger project concerned with the general demand-driven model checking of symbolic systems through the use of temporal and stochastic logics.

This paper is organised in the following way: Binary Decision Diagrams are introduced in Section 2.1 and two pre-existing methods for creating them are presented in Sections 2.2 and 2.3. We describe our lazy algorithm in Section 3. In Section 4, we look at a specific model checking technique that benefits from lazy BDD evaluation and further discuss how the existing methods would have coped with this model checking method (Section 4.3). The paper concludes in Section 5 with a discussion of current and future work and applications to other symbolic model checking methods.

2 Binary Decision Diagrams

2.1 Summary

Binary Decision Diagrams are a graph representation of If-then-else Normal Form[1] (INF). For a given Boolean expression and variable order, there is a unique minimal BDD representation (Bryant 1986 [4]). BDDs are often substantially more compact than DNF (sum-of-products) or CNF (product-of-sums) and have proved very successful in symbolic representation for computer-aided verification (Burch et al 1990a [7], Burch et al 1990b [6], Dill 1993 [10]).

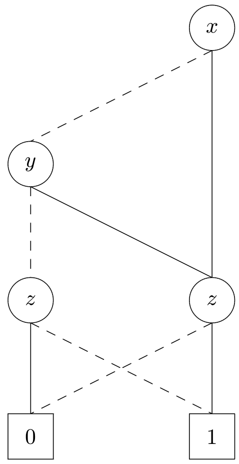

Figure 1: An example of a BDD for $(x \vee y) \Leftrightarrow z$ (Dashed lines are 0-edges, solid lines are 1-edges)

In structure, BDDs are essentially reduced binary decision trees (similar to *function graphs* in Bryant 1986 [4]), except that nodes that represent the same function are unified to form a directed acyclic graph or DAG (Figure 1). For any BDD there is also a strict total order on the evaluation of variables as the graph is created from root to terminal node[2]. The space complexity for an expression depends intrinsically on the order in which the variables are evaluated (Bryant 1986 [4]). For many Boolean functions with an optimal variable ordering, the size of a BDD is $O(n)$ nodes for n variables (Akers 1978 [1]). However in the worst case a BDD can have $O(2^n/n)$ nodes (Akers 1978 [1]), which is still an improvement on the binary decision tree's $O(2^n)$ space usage.

There are two main methods of variable-based BDD construction. That presented in Bryant 1986 [4] and an efficient method presented in Brace et al 1990 [3]. Both concentrate on the variable ordering of a BDD to construct a Reduced Ordered BDD from the outset. We present a summary of both methods here so that comparisons can be made between the operation of these methods and the lazy algorithm presented later.

[1] If-then-else (ite) is a ternary Boolean operator such that: $ite(p, q, r) \equiv (p \rightarrow q, r) \equiv (p \wedge q) \vee (\neg p \wedge r)$

[2] In this form, they are also known as Reduced Ordered BDDs (ROBDDs) or just Ordered BDDs (OBDDs).

2.2 Traditional Method

Traditionally BDDs have been implemented as a sequential array of triples `(var, lo_exp, hi_exp)`, where the `var` identifies the variable in the Boolean expression. The `lo_exp` and `hi_exp` identifiers are integer pointers to other triples in the array, representing the zero and one edges from that node.

In this approach, construction of the BDD uses Shannon decomposition on each variable of the Boolean expression:

$$t = (x \to t[1/x], t[0/x])^3$$

This method calculates the *function graph* for the expression before reducing it to the BDD. On each pass of the algorithm, the next variable in the order is used to decompose the expression recursively until a terminal Boolean value is reached. At this stage, the recursion unwinds and the reduced and ordered BDD is created from the bottom up, starting from terminal nodes.

Since two sub-expressions are generated for each instance of a variable, the algorithm generates an exponential number of sub-expressions (2^n for n variables) in memory before any space reduction to the BDD form can take place. The reason being that this stage represents the creation of the function graph, which is invariably exponentially large. Bryant 1986 [4] presents the second stage of the algorithm which reduces a function graph to a BDD but other implementations (Andersen 1996 [2]) explicitly produce the function graph in this way.

There are also other and more fundamental side effects caused by selecting one variable and forcing evaluation of that variable throughout a Boolean expression. As we will see in Section 4.3 this impedes lazy (or non-strict) evaluation from the symbolic representation level.

2.3 Efficient Method

The efficient implementation of a BDD package presented in Brace et al 1990 [3] is a reapplication of the Shannon decomposition method outlined above, which gets round some of the memory problems associated with an exponential function graph. It creates a hash table (or *unique table*) of `(var, hi_exp, lo_exp)` which represents the equivalent *ite* (if-then-else form) expression.

As before, the `lo_exp` and `hi_exp` identifiers are pointers to other nodes in the hash table. The main difference from the original method is one of representation. The original assignment method only allows the representation of a single Boolean function, whereas the *unique table* here allows the representation of many Boolean functions at once using a multi-rooted BDD. This also means that expressions with similar sub-structures actually share nodes in the *unique table*. This enhancement requires the introduction of node level garbage collection so that nodes are removed if and only if they are not needed by any of their parent functions.

The main computational improvement gained from using a multi-function representation is that identical functions have to share the same *unique table* entry thus making BDD comparison a constant time operation.

Another implementation improvement is the use of a memory function (or *memo-izing*) through a *computed-table* to perform quick lookup on formulae that have already been converted to BDDs. This means that repeating Boolean formulae do not have to be recalculated.

This method uses the fact that a general expression Z has the form:

$$Z = (F \to G, H)$$

where F, G, H are in ROBDD form and v is the first variable for each. The method then uses Shannon to decompose Z recursively:

$$\begin{aligned} Z \quad = \quad & (v \to (F[1/v] \to G[1/v], H[1/v]), \\ & (F[0/v] \to G[0/v], H[0/v])) \end{aligned} \tag{1}$$

Since v is at the head of each term F, G, H, this does not require substituting and re-evaluating the expression (as with the original method) but is simply a matter of extracting the correct branch from

[3] $t[p/q]$ represents the expression t with every occurrence of q replaced with p.

the head of the respective BDD. Although no explicit assignments are performed, the necessity for F, G, H to be in strict variable order means that either they must be coerced into that order or evaluated in a bottom-up fashion by recursive application of the algorithm. This is a common feature with the original method described in Section 2.2.

The strict predefined variable ordering of this method and the required bottom-up evaluation negate the apparent compositionality of Equation (1) and mean that the algorithm is unsuitable for lazy evaluation. All the constituent BDDs must exist in the *unique table* before the parent BDD can be created—this dependency forces strict evaluation.

3 Lazy BDD Construction

3.1 Introduction

In this method, rather than concentrate our attention on the variables in a given Boolean function, we use the symbolic structure of the expression itself to generate the BDD.

the fact that we already know the BDD for the expression $x_1 \wedge x_2$, for example, can be used as an initial step to evaluate the BDD for the compound expression $t_1 \wedge t_2$. This is achieved by a process of efficient composition. So for some general binary operator ($\circ$) such that $f(p, q) = p \circ q$:

$$\Rightarrow \quad f(t_1, t_2) = f(x_1, x_2)[t_1/x_1, t_2/x_2]$$

We further generalise this by expressing all the usual Boolean operators in *ite* form (as in Figure 1 from Bryant 1986 [4]) and then using the known BDD structure for $(t_1 \to t_2, t_3)$.

3.2 Description

At each stage of the generating process the top level Boolean operator is removed and the BDD is updated with the BDD representation of that operator. The sub-expressions of that operator can then be treated as if they were atomic variables (similar to the *If-then-else DAGs* in Karplus 1989 [11]). As such, they can be assigned individual nodes with zero and one edges until further decomposition is required and the process is repeated recursively.

For complete evaluation of a BDD all such child expressions are decomposed until actual variable nodes are obtained throughout the BDD.

If $t = (t_1 \to t_2, t_3)$ represents the decomposition of the expression t into *ite* form then: $t = (x \to t_2, t_3)$ (for a variable x) is the terminating condition for the removal of any non-variable expressions from a given node.

Use of operator symmetry is important for ensuring that variables occupy early nodes as soon as possible in the decomposition process. This ensures that unevaluated expression nodes are encountered as late as possible and partial evaluation is optimally used. So in the case of $\wedge$ and $\vee$: $t = (t_1 \to 1, x) = (x \to 1, t_1)$ and $t = (t_1 \to x, 0) = (x \to t_1, 0)$.

To achieve the efficient composition process mentioned in Section 3.1, the actual data structures used are similar to Brace et al 1990 [3]. Indeed, the method employs a similar *unique table* data structure in the form of a hash table, rather than the indexed array used in the original method. The algorithm itself consists of an initial construction function and a node decomposition function, which is called until a variable node is exposed.

Lazy or demand-driven evaluation is achieved by only calling `decompose_node` (in Figure 3.2) when that node is required by an application process.

As in Brace et al 1990 [3], similar *ite* rewriting rules apply to keep the BDD in a reduced form: $t = (0 \to t_2, t_3) = (t_3 \to 1, 0)$ and $t = (1 \to t_2, t_3) = (t_2 \to 1, 0)$

The memory requirement for this basic method is $O(|t|)$, where $|t|$ is the the number of nodes in the BDD form of expression t. Further, we can say that the memory complexity in building the BDD never exceeds the final size of the BDD. This is in contrast to the traditional method which, while calculating the BDD, will certainly exceed the memory needed to store the BDD.

```
bdd_construct(t) {
   id = insert_hashtable(t, 1, 0);
   return id;
}

decompose_node(id) {
   if (id.expression is variable)
      return id;

   ite(t1, t2, t3) = id.expression;
   r_hi = insert_hashtable(t2, id.hi, id.lo);
   r_lo = insert_hashtable(t3, id.hi, id.lo);
   id = update_hashtable(id, t1, r_hi, r_lo);
   return id;
}
```

Figure 2: Lazy evaluation algorithm

3.3 Canonicity and Variable Order

In Brace et al 1990 [3], canonicity is assured by a hash-value label assigned by the *unique table* and by the fact that evaluation takes place from the bottom-up in order to maintain variable ordering. Since our method uses a top-down evaluation technique (which is essential for a lazy approach), we do not have the luxury of knowing whether two nodes are identical in the same way.

The lack of automatic node canonicity means that we have to put more work into assuring that all the available sharing is found. Adopting the approach of Brace et al 1990 [3] would destroy the demand-driven evaluation that we are seeking. Our current approach is to ensure as much canonicity (and thus graph sharing) as the current partially evaluated state of the BDD will allow.

Let us define the concept of *stable* nodes recursively:

i. An evaluated variable node is *stable* if both of its child nodes are *stable*

ii. An unevaluated expression node is not *stable*

iii. The terminal nodes are *stable*

The notion of *stability* comes from the fact that an expression node will have a different hash value when it has been decomposed. Therefore potentially every parent node will be affected in the same way.

An expression node becomes *stable* when it is decomposed into two *stable* variable nodes. Only at this point is it worth recalculating the hash values of all the parent nodes so that strong canonicity can be enforced.

We achieve this by using temporary *pointer nodes* to act as indirection to maintain the correctness of parent node references. The parent nodes are updated and all indirection and *pointer nodes* are removed during the garbage collection process [4].

When an expression becomes *stable* through decomposition, it is rehashed to a new position in the *unique table* and a *pointer node* is placed in the old position. Subsequently as the BDD is traversed in normal operation parent nodes are updated with the new hash value and they in turn are rehashed and *pointer nodes* are used as indirection for their parent nodes.

As *pointer nodes* are dereferenced by their parent nodes they can be garbage collected in the same way as normal nodes. Alternatively, on rehashing to a larger *unique table*, all the pointer nodes can be dereferenced and removed at very little additional cost [5].

When a root node becomes *stable*, it can be compared to other Boolean functions in the standard way as long as variable ordering has been maintained. However, the algorithm as it stands will not combine repeated variables as they are resolved from unevaluated expressions. Variable ordering is therefore enforced periodically by the coercion of the partially evaluated BDD into a prespecified variable order, using the standard techniques (Bryant 1986 [4]). This is a space-for-time tradeoff; we benefit from the space saving of having a partially evaluated BDD, but we pay for this by having to coerce the variable order every so often (an $O(|t|\log|t|)$ activity). Care should be taken at this stage since optimal orderings for the fully evaluated

[4]Identical to the standard garbage collection method used in Brace et al 1990 [3], except extended to include *pointer nodes* as well.

[5]A similar method is used in dynamic memory management systems.

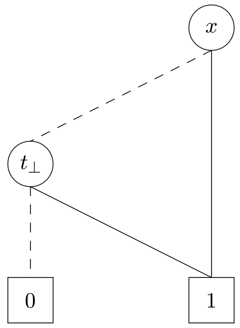

Figure 3: Partially evaluated BDD $x \vee t_\perp$

BDD may not necessarily be optimal for partially evaluated BDDs—this is the subject of further research.

3.4 Benefits of Lazy Evaluation

The interest in this method lies in the realisation that useful BDDs can be generated while the generating Boolean expression is still only partly decomposed. We need to define what a "useful BDD" is for our purposes.

A useful BDD is one that contains 0 and 1 nodes and a root variable and at least one path to either of the terminal nodes which does not go via an undecomposed expression node. This is exactly the situation we need to exploit lazy partial evaluation of BDDs and thus reduce computational and storage requirement.

This situation is shown in Figure 3 which might show the first part of a long disjunction. In its current state, the BDD currently represents $x \vee t_\perp$, for some variable x and some as yet unevaluated expression $t_\perp$.

Of course this is dependent on the evaluation requirement asked of the BDD. If the BDD is reduced along a path that leads to a terminal 0 or 1 node, then it is successful. Otherwise it will need to partially decompose the Boolean expressions further to see if a path to the terminal nodes can be found. However the model checking requirement (or whatever the application driving the decomposition) might still force the evaluation of the entire expression.

So in the example, Figure 3, if it is known that $[1/x]$ then the BDD can be evaluated without ever evaluating $t_\perp$. Or in reverse, if a path to the 1 node is required (satisfiability) then similarly this can be provided without decomposing $t_\perp$ any further. However there may be a requirement which involves $[0/x]$ or more than one satisfiability path, in which case $t_\perp$ will need to be decomposed at least one step more.

Symbolic representations have alternative DNF expressions which are particularly amenable to partial BDD evaluation, since each clause of the DNF has a path to the 1-node which bypasses further evaluation of subsequent clauses. A useful heuristic, as far as is possible, is that clauses representing important states should be encoded first so that they are accessible first when the BDD is partially evaluated. The definition of important is very much dependent on the model checking application being run.

4 Model Checking

4.1 Summary

Our direct interest in BDDs lies in the efficient representation of state space for symbolic model checking. With this in mind, we are also interested in how particular symbolic system descriptions are represented as Boolean expressions and how these might benefit from lazy BDD evaluation.

The method focussed on here involves creating a transition relation or characteristic function for a particular system. Using symbolic reachability (Dill 1993 [10], Andersen 1996 [2]) we can construct the entire set of reachable states for a system and then perform model checking tests on the set.

The major problem with evaluating the reachable states is that inevitably it suffers from a state-space explosion for systems of any complexity (Bryant 1992 [5]). A lazy method for generating and storing the reachable states clearly has the potential to prevent or at least delay such an explosion long enough for useful model checking information to be gathered.

4.2 Symbolic Reachability

In a finite system, states are represented by n Boolean variables and a set of such states can typically be represented by a DNF of these variables.

Traditionally the whole reachable state-space has to be evaluated before model checking can proceed. This is clearly a waste of resources if not all the states are required for the model checking process. Using lazy BDD evaluation we can potentially generate satisfiable paths as they are required by the model checking procedure. This is especially applicable to the generation of counter-examples or *witnesses* to particular model checking properties (Clarke et al 1994 [9])[6].

In model checking terms: the symbolic reachability technique operates on a predicate of these n variables to generate a fixed point:

$$\mu R. I \vee (\exists \vec{x}.\ T \wedge R)[\vec{y}/\vec{x}],$$

which represents the total reachable state-space. In practice, this is calculated using the iterative formula:

$$R_{i+1}(\vec{x}) = I \vee (\exists \vec{x}.\ T(\vec{x}; \vec{y}) \wedge R_i(\vec{x}))[\vec{y}/\vec{x}],$$

where I is the initial state, T is the transition function from state $\vec{x}$ to state $\vec{y}$ and R_i is the ith iteration of the reachable states expression. The iteration terminates when $R_{i+1} = R_i$.

4.3 Comparison of Methods

In implementing symbolic reachability, there is a choice of representation of the various functions I, T, R. They can be calculated symbolically using the higher level Boolean operators with the resulting DNF expression converted into a BDD. Alternatively they can be stored as BDDs from the start and the Boolean operators converted into BDD operations.

It is at this stage that it can be seen why the two original methods (presented in Sections 2.2 and 2.3) would be inappropriate for lazily evaluating large DNF equations in a clause-by-clause fashion.

The traditional evaluation technique (Section 2.2) requires each variable to be assigned a value at each node, so if that variable exists in every part of the disjunction, then the whole disjunction must be known in order for the assignment to take place.

In the efficient method (Section 2.3), the variable order requirement means that the whole expression would have to have been evaluated from the bottom up in order to have achieved the correct variable order at each stage. Similarly the whole disjunction must exist for this to occur and no element of lazy evaluation can be employed.

For both strict methods, maintaining I, T, R as BDDs throughout the calculation is therefore the only option. The end result may then itself be an exponentially-sized BDD (which is likely for complex systems with many states), so making it impossible to maintain a fully evaluated version, far less perform model checking on it.

From Section 3.4, we know that our lazy technique can do a clause-by-clause conversion from the DNF. The DNF expression itself can then be generated lazily, which is essential for memory reasons. Alternatively it can act as the expression representation for I, T, R from the start and maintain a usable representation for as long as the model checking requirement will allow.

5 Conclusions

We have presented an efficient algorithm for the lazy partial evaluation of Binary Decision Diagrams. The space and time complexity of the lazy algorithm has been compared with the traditional techniques and it has been shown that for applications requiring only partial evaluation of the BDD (such as in Clarke et al 1994 [9]), significant space gains can be made. We are currently in the process of applying the algorithm to a series of test cases. These will allow us to present some quantitative data for the execution speed, space used and, importantly, percentage of total BDD expanded, under specific model checking requirements.

[6] These are just satisfiable paths of the negated model checking property.

It is realised that our lazy technique, in common with any lazy method, is not optimal when complete evaluation is required from the outset. This means that certain model checking tasks, such as state-wise equivalence, will always be better accomplished by a method such as Brace et al 1990 [3].

In this paper, we have looked at how this method benefits one form of model checking: reachable state analysis and in particular the generation of counter-examples (Clarke et al [9]). Future research will show whether this lazy method could be applied to the *iterative squaring* technique. This has particular application to generalised symbolic model checking through the implementation of a Mu-Calculus (such as described in Stirling 1991 [12]), pioneered by Burch et al [7]. There appears to be significant potential for lazy evaluation in this area since the transitive closures involved are often extremely large (Dill 1993 [10]) and therefore impractical under strict evaluation.

References

[1] Akers, S. B. Binary Decision Diagrams. *IEEE Transactions on Computers C-27*, 6 (June 1978), 509–516.

[2] Andersen, H. R. An introduction to Binary Decision Diagrams. Course notes, Deptartment of Computer Science, Technical University Denmark, September 1996.

[3] Brace, K. S., Bryant, R. E., and Ruddell, R. L. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, June 1990), ACM, pp. 40–45.

[4] Bryant, R. E. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (August 1986), 677–691.

[5] Bryant, R. E. Symbolic Boolean manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys 24*, 3 (September 1992), 293–318.

[6] Burch, J. R., Clarke, E. M., McMillan, K. L., and Dill, D. L. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, June 1990), ACM, pp. 46–51.

[7] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. Symbolic model checking: 10^{20} states and beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Science* (Philadelphia, June 1990), IEEE, pp. 428–439.

[8] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8*, 2 (April 1986), 244–263.

[9] Clarke, E. M., Grumberg, O., McMillan, K., and Zhao, X. Efficient counterexamples and witnesses in symbolic model checking. Tech. Rep. CMU–CS–94–204, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, October 1994.

[10] Dill, D. Automatic formal verification of finite-state systems. Workshop on logical methods in concurrency, Department of Computer Science, Aarhus University, DK–8000 Aarhus C, Denmark, August 1993.

[11] Karplus, K. Using if-then-else DAGs for multi-level logic minimization. In *Proceedings of the 6th MIT Conference on Advanced Research in VLSI* (Cambridge, Mass., 1989), C. Seitz, Ed., MIT, MIT Press, pp. 101–118.

[12] Stirling, C. An introduction to modal and temporal logics for CCS. In *Proceedings of the 1989 UK/Japan Workshop on Concurrency: Theory, Language and Architecture* (Oxford, September 1989), A. Yonezawa and T. Ito, Eds., vol. 491 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 2–20.

[13] Wilf, H. S. *Algorithms and Complexity*. PHI Editions. Prentice Hall, 1986.