# Specifying Security Constraints with Relaxation Lattices

Maurice P. Herlihy and Jeannette M. Wing
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

## 1. Introduction

Large software systems used in practice typically exhibit more complex behavior than small well-understood programs. Often, such systems display degraded behavior as they react to changes in the environment. Under ideal circumstances, the system's behavior satisfies a set of application-dependent *preferred constraints.* Each constraint typically preserves a certain level of "correctness," and each has an associated cost. In the presence of events not under the system's control, e.g., faults due to its environment, certain constraints may become difficult or impossible to satisfy, and the application designer may choose to relax them as long as the resulting behavior is sufficiently "close" to the preferred behavior. Other external events, e.g., bug fixes or compensating actions, can later cause a system to return to a more preferred behavior. In the security application, faults correspond to breaches in security and integrity, e.g., students modifying a grades file; fixes correspond to repairing security holes, e.g., changing the passwords of accounts that have been illegally accessed.

Numerous specification techniques have been successfully used to characterize the functional properties, i.e., input-output behavior, of systems but only a few have been applied, with limited success, to specify security properties as well (e.g., [2, 6, 12, 7]). These attempts typically treat security constraints as additional functional constraints of a system, resulting in a monolithic specification that fails to distinguish between correctness properties independent of security concerns and those specific to privacy and protection.

In this paper, we describe the *relaxation lattice method*, a new approach to specifying graceful degradation for a large class of systems. We apply this method to the security domain by identifying degraded system behaviors with those that may result from security violations such as a user of one security class obtaining access rights associated with those of a higher class. Our method can be used in two ways: (1) as a descriptive technique for specifying the behavior of existing systems in which breaches of security may inadvertently or unavoidably occur and (2) as a formal design technique for specifying a range of behaviors, from ideal to undesired, of systems to be implemented.

The key to our method is the incorporation of *sets of constraints* into standard (functional) specifications. As with the usual correspondence between specifications and implementations, the less constraining the specification, the greater the number of possible implementations. The significant advantage our method enjoys over others is the clean separation between the specification of a system's functional behavior in the absence of faults and that in the presence of faults due to its environment. By factoring out correctness constraints from security constraints, we can characterize the essential trade-offs between the costs of preserving security constraints and the costs of relaxing them. Thus, the price one must pay for a secure system can be calculated in terms of the level of security desired.

## 2. General Model

The basic containers for data are called *objects*. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to create and manipulate objects of that type. For example, a file might provide Read and Write operations, and a FIFO queue might provide Enq and Deq operations.

We model a computation as a *history*, which is a finite sequence of executions of operations on objects. For an operation executed by a process *p* on an object *x* in a history, we write:

$$x :: op(args^*)/term(res^*) :: P$$

where *op* is an operation name, *args** is a sequence of argument values, *term* is a termination condition name, and *res** is a sequence of result values. The operation name and argument values constitute the *invocation*, and the termination condition and result values constitute the *response*. We use "Ok" for normal termination. We assume that operations on objects can be executed atomically; that is, an operation either takes place completely or not at all, and operations appear to take place instantaneously with respect to one another.[1] An *object subhistory*, H|x, of a history H is the subsequence of operations in H whose object names are x. In this paper, we focus on individual object subhistories.

### 2.1. Simple Object Automata

We model an object by a *simple object automaton*, an automaton that accepts certain sequences of operation executions. A simple object automaton is a four-tuple $\langle \text{STATE}, s_0, \text{OP}, \delta \rangle$, where STATE is the object's set of states, $s_0 \in \text{STATE}$ is its initial state, OP is a set of operations (the automaton's input alphabet), and $\delta: \text{STATE} \times \text{OP} \rightarrow 2^{\text{STATE}}$ is a partial *transition function*.

[1]Atomic operations can be implemented by a variety of well-known techniques, including the two-phase locking and two-phase commitment protocols [5, 8], or atomic broadcast protocols [3, 4].

The domain of the transition function can be extended to histories, $\delta^*: \text{STATE} \times \text{OP}^* \rightarrow 2^{\text{STATE}}$:

$$\delta^*(s, \Lambda) = s$$

$$\delta^*(s, H \cdot p) = \cup_{s' \in \delta^*(s,H)} \delta(s', p)$$

where "$\cdot$" denotes concatenation, and "$\Lambda$" denotes the empty history. We use $\delta^*(H)$ as shorthand for $\delta^*(s_0, H)$. A history H is *accepted* by an automaton if $\delta^*(H) \neq \emptyset$. We call $\boldsymbol{L}(A)$, the language accepted by automaton A, the *behavior* of A.

### 2.2. Relaxation Lattices

Let $\boldsymbol{A}$ be a set of simple object automata having the same set of states, the same initial state, and the same operations, but (possibly) different transition functions. We say that $\boldsymbol{A}$ is a *lattice of automata* if the set $\{\boldsymbol{L}(A) \mid A \in \boldsymbol{A}\}$ is a lattice under reverse inclusion (i.e., the smallest language is at the top). We call the language of the automaton at the top of the lattice the *preferred behavior* of the lattice.

A *relaxation lattice* is given by a set of constraints $\boldsymbol{C}$, a lattice of automata $\boldsymbol{A}$, and a lattice homomorphism, $\phi: 2^{\boldsymbol{C}} \rightarrow \boldsymbol{A}$. For now, we leave a relaxation lattice's set of constraints uninterpreted since the meaning of such constraints is domain-dependent. It suffices to think of each constraint as an assertion to be satisfied. We will see that in the security domain, the set of constraints roughly corresponds to the complement of the set of capabilities that processes have with respect to objects (protected resources) in the system. We orient the lattice $2^{\boldsymbol{C}}$ so that the largest (intuitively, the strongest) set of constraints lies at the top, and $\phi(\boldsymbol{C})$ is the preferred behavior of $\boldsymbol{A}$. In general, $\phi$ is defined over a sublattice of $2^{\boldsymbol{C}}$.

A relaxation lattice is thus a lattice of simple object automata parameterized by a set of constraints, where the stronger the set of constraints, the smaller the language accepted. Informally, a relaxation lattice describes an object's *conditional* behavior. If the environment is such that the object satisfies constraints $C \subseteq \boldsymbol{C}$, then the object will behave like the simple object $\phi(C)$, accepting the language $\mathbf{L}(\phi(C))$. While an object is able to satisfy its strongest set of constraints, it will accept only histories from its preferred behavior. If changes to the environment, e.g., security violations, force the object to satisfy a weaker set, then it will accept additional "weakly correct" histories, which are undesired but perhaps tolerated. Further changes to the environment may later cause the object to satisfy a more desired behavior.

The relaxation method is appropriate for modeling the behavior of objects for which there is a meaningful cost associated with moving up the relaxation lattice. The higher one goes in the lattice, the higher the price paid for the more preferred behavior. In the security domain, we use constraints to model the cost of tolerating violations such as the cost of implementing a secure encryption algorithm or the cost of hiring personnel to guard a locked room.

## 2.3. The Environment

The environment determines which behavior, preferred or otherwise, an object exhibits. The environment itself can be represented by an automaton $<2^{\mathbf{C}}, c_0, \text{EVENT}, \delta_E>$, where input events in EVENT model changes in the current set of constraints (state), and $\delta_E$: $2^{\mathbf{C}} \times \text{EVENT} \rightarrow 2^{\mathbf{C}}$ is the transition function (note that $\delta_E$ maps to a single state, not a set of states as for object automata). Let $\mathbf{A}$ be a lattice of automata, where each A in $\mathbf{A}$ is given by the tuple $<\text{STATE}, s_0, \text{OP}, \delta_A>$. The sets EVENT and OP may be disjoint, as in the mail queue example of the next section, but in general they may overlap. Let $\phi$: $2^{\mathbf{C}} \rightarrow \mathbf{A}$ be the lattice homomorphism.

The environment and the lattice can be combined into a single automaton that accepts interleaved events and operations:

$$<2^{\mathbf{C}} \times \text{STATE}, (c_0,s_0), \text{EVENT} \cup \text{OP}, \delta>$$

Let EVENTOP be EVENT $\cup$ OP. The transition function $\delta$: $2^{\mathbf{C}} \times \text{STATE} \times \text{EVENTOP} \rightarrow 2^{\mathbf{C}} \times 2^{\text{STATE}}$ is defined by two components, $\delta_1$: $2^{\mathbf{C}} \times \text{EVENTOP} \rightarrow 2^{\mathbf{C}}$, which defines the effects on the environment state, and $\delta_2$: $2^{\mathbf{C}} \times \text{STATE} \times \text{EVENTOP} \rightarrow 2^{\text{STATE}}$, which defines the effects on the lattice state:

$$\delta_1(c, p) = \text{if } p \in \text{EVENT then } \delta_E(c, p) \text{ else } c$$

$$\delta_2(c, s, p) = \text{if } p \in \text{OP} \wedge A = \phi(\delta_1(c, p)) \text{ then } \delta_A(s, p) \text{ else } \{s\}$$

When the (combined) automaton accepts an event, it changes the environment state. When the automaton accepts an operation, it changes the object state, choosing the transition function indicated by the current environment. If the input is both an event and an operation, the environment changes before the transition function is selected.

## 2.4. Specification Language

In our examples, we use the Larch Specification Language [9] to specify the automata of the lattice $\mathbf{A}^2$, in particular, the STATE and $\delta$ components of a simple object automaton. A state in STATE is a mapping between an object and its *value*, hence it is convenient to represent an object's possible states as a set of values. We specify an object's values with a Larch *trait*, which denotes a first-order theory. In a trait, the set of operators and their signatures following **introduces** defines a vocabulary of terms to denote values. For example, from the Queue trait of Figure 2-1, *emp* and *ins(emp, 5)* denote two different queue values. The set of equational axioms following the **constrains** clause defines a meaning for the terms, more precisely, an equivalence relation on the terms, and hence on the values they denote. For example, from Queue, one could prove that *del(ins(ins(ins(emp, 4), 3), 3), 3) = ins(ins(emp, 4), 3)*. The **generated by** clause of Queue asserts that *emp* and *ins* are sufficient operators to generate all values of queues. Formally, it introduces an inductive rule of inference that allows one to prove properties of all terms of sort *Q*.

We use Larch *interfaces* to describe transition functions for simple object automata. For example, interfaces for the Enq and Deq operations for queues are shown in

[2] We use informal descriptions to characterize the environment, though it is straightforward to give it a complete Larch specification since an environment is also modeled as an automaton.

Figure 2-2. The object's identifier, e.g., q, is an implicit argument and return formal of each operation; the process's (immutable) identifier, e.g., P, is an implicit argument. A **requires** clause states the precondition that must hold when an operation is invoked. An **ensures** clause states the postcondition that the operation must establish upon termination. An unprimed argument formal, e.g., q, in a predicate stands for the value of the object when the operation begins. A return formal or a primed argument formal, e.g., q', stands for the value of the object at the end of the operation. For an object *x*, the absence of the assertion $x' = x$ in the postcondition states that the object's value may change. We use the vocabulary of traits to write the assertions in the pre- and postconditions of an object's operations; we use the meaning of equality to reason about its values. Hence, the meaning of *ins* and = in Enq's postcondition is given by the Queue trait. Note that by definition of the trait operators, the Deq operation does not necessarily remove

```
Queue: trait
 introduces
   emp: → Q
   ins: Q, E → Q
   del: Q, E → Q
   isEmp: Q → Bool
   isIn: Q, E → Bool
 constrains Q so that for all [q: Q, e, e1: E]
   Q generated by [ emp, ins ]
    del(emp, e) = emp
    del(ins(q, e), e1) = if e = e1 then q
                          else ins(del(q, e1), e)
    isEmp(emp) = true
    isEmp(ins(q, e)) = false
    isIn(emp, e) = false
    isIn(ins(q, e), e1) = (e = e1) ∨ isIn(q, e1)
```

**Figure 2-1:** Queue Trait

```
q:: Enq(e)/Ok() ::P
   requires true
   ensures q' = ins(q, e)

q:: Deq()/Ok(e) ::P
   requires ¬ isEmp(q)
   ensures isIn(q, e) ∧ q' = del(q, e)
```

**Figure 2-2:** Queue Interfaces

the first element inserted as for a FIFO queue, and permits for duplicate elements as for a multiset.

For an operation, *p*, of a simple object automaton, A, we write $p.pre_A$ and $p.post_A$ for the pre- and postconditions of *p*. The transition function δ for A is defined such that

$$(\forall\, s, s' \in \text{STATE})\ s' \in \delta(s, p) \quad \text{iff } p.pre_A(s) \wedge p.post_A(s, s').$$

For each automaton in a lattice ***A***, the sets of states (values) are the same, but their transition functions differ. Thus, their specifications will all use the same trait, but will have different interfaces.

## 3. Application of Model to Security

Relaxation lattices can be used to specify certain kinds of security properties. Informally in this domain, the objects are the resources to be protected, e.g., files, directories, and laserwriters, and the processes are the users, e.g., people and programs, of these resources. The environment captures the privileges of the users, i.e., the rights of users to execute certain operations on the resources. The environment corresponds to an access-rights matrix [11], which can change as protection is breached. To preserve *secrecy*, we must ensure that unauthorized users are prevented from executing operations that return information about the object's state, and to preserve *integrity*, we must ensure that unauthorized users are prevented from executing operations that modify the object's state.

### 3.1. Secure Object Automata

Let *S* and *O* be the sets of subjects and objects. Intuitively, *S* consists of all system users and programs, i.e., processes in our general model; *O* consists of all the resources to be protected, e.g., files, directories, and laserwriters. Let *M* be an *access-rights matrix* [11] where the $(i, j)$-th entry in *M* is a set of rights that subject $i \in S$ has for object $j \in O$.

Unlike standard models of security (e.g., Bell and LaPadula's [1] or Lampson's [11]) in our model, a *right* of a subject *i* is not just the name of an access mode (e.g., Read, Modify, Execute) or operation (e.g., Enq, Deq), but is a pair of predicates (i.e., pre- and postconditions) on the name of each operation of object *j*. For example, an entry for a subject P on a file f might contain the following pair of predicates for a Write operation:

```
f:: Write(v: value) :: P
   requires id(P) = owner(f)
   ensures val(f') = v
```

where "id", "owner," and "val" are defined in the appropriate traits. This element of the (P, f) entry in *M* restricts the process P invoking the Write operation to be the owner of the file f.

> **Definition 1:** A history H is *secure* if for each operation "A:: e ::P" in H there exists some (pre, post) pair of predicates for operation e in the (P, A)-th entry of *M* such that e.pre(s) $\wedge$ e.post(s, s'), where s, s' $\in$ $\text{STATE}_A$ and s is the state of A upon invoking e and s' is the state upon return.

A *secure object automaton* Secure(A) is an object automaton that accepts histories of the simple object automaton A such that each history in *L*(Secure(A)) is secure.

Since an access-rights matrix *M* can be viewed as a set of permissions, we identify an environment's set of constraints (its "state") to be the complement of the set of permissions. Intuitively, constraints are prohibitions of the form "User X does not have the capability for operation Y on object Z," that are formally derivable from the sets of pairs of predicates of *M*.

## 3.2. Secure Mail Queue

Although secrecy and integrity are often treated as monolithic properties, they can be viewed as subject to graceful degradation. For example, consider a *mail queue* used as a temporary repository for mail intended for other sites. The mail queue provides four operations: a user can enqueue a message, dequeue the oldest message from the queue, cancel an unsent message, and list unsent messages. Clearly, not everyone should be allowed to execute every operation. For this example, we recognize four disjoint classes of users (see 3-1): (1) *operators* may execute any operation, (2) *faculty* members may enqueue messages and list or cancel transmission of their own messages, (3) *mailer* processes may dequeue messages for transmission, and (4) *students*, naturally, have no privileges at all. The specification of the set of values (hence, STATE) of the mail queue is given in Figure 2-1, and of the operations (hence, the transition function $\delta$), in Figure 3-2. In this specification, we assume that each user U invoking an operation on the queue q has an unforgeable name id(U), and that any attempt to execute an unauthorized operation signals an *Unauthorized* exception.

```
Class: trait
  C enumeration of
    [operator, faculty, mailer, student]

Users: trait
  includes Class, Set[USet, User]
    User record of [id: Id, class: C]
  introduces
    ops: USet → USet
    fac: USet → USet
    mlr: USet → USet
    stu: USet → USet
  constrains USet so that for all [u: U, us: US]
    u ∈ ops(us) = [class(u) = operator ∧
        u ∉ (fac(us) ∪ mlr(us) ∪ stu(us))]
    u ∈ fac(us) = [class(u) = faculty ∧
        u ∉ (ops(us) ∪ mlr(us) ∪ stu(us))]
    u ∈ mlr(us) = [class(u) = mailer ∧
        u ∉ (ops(us) ∪ fac(us) ∪ stu(us))]
    u ∈ stu(us) = [class(u) = student ∧
        u ∉ (ops(us) ∪ fac(us) ∪ mlr(us))]
```

**Figure 3-1:** Traits for Users of the Secure Mail Queue

We can use relaxation lattices to formulate a variety of alternative "less secure" mail queue specifications. We take as our lattice of constraints prohibitions of the form "User X does not have a capability for operation Y." Under ideal circumstances, each user has the set of capabilities appropriate to his or her class. The cost of preserving these constraints is the cost of keeping passwords secret, using secure encryption protocols, etc. An event that affects the environment occurs when a user acquires capabilities to which he or she is not entitled, increasing the set of possible behaviors. Since a user who has discovered a new security breach is always free to refrain from exploiting it, each such breach introduces the possibility of new behaviors without excluding the possibility of older behaviors, thus the resulting set of behaviors clearly forms a relaxation lattice. The relaxation lattice characterizes the extent to which the resulting insecure behavior is close to the preferred secure behavior.

Suppose Alice is a faculty member and Bob a student. Ideally, the following constraints hold (among many others):

| | |
|---|---|
| S1 | Alice cannot dequeue messages. |
| S2 | Bob cannot list Alice's message. |

If Alice discovers a way to relax constraint S1, then the specification for Deq would change as shown in Figure 3-3. Note that because Alice is always free to refrain from exploiting her knowledge, the precondition for the *Unauthorized* signal remains unchanged, and the corresponding automaton accepts a strictly larger language. Note also that the specification is independent of *how* Alice manages to circumvent the prohibition against dequeuing messages. Similarly, Bob might relax constraint S2 if he learns Alice's password. The resulting specification for List is shown in Figure 3-4. Here too, since Bob can always refrain from using Alice's password, the precondition for the *Unauthorized* exception is unchanged, and the set of possible behaviors is strictly larger. Finally, an even larger set of behaviors is possible if both constraints are relaxed.

```
q:: Enq(m: message) / Ok() ::U
  requires class(U) = operator ∨
           class(U) = faculty
  ensures q' = ins(q, m)

q:: Enq(m: message) / Unauthorized() ::U
  requires ¬(class(U) = operator ∨
             class(U) = faculty)
  ensures q' = q

q:: Deq() / Ok(m: message) ::U
  requires class(U) = mailer ∨
           class(U) = operator
  ensures q' = del(q, m)

q:: Deq() / Unauthorized() ::U
  requires ¬(class(U) = mailer
             ∨ class(U) = operator)
  ensures q' = q

q:: Cancel(m: message) / Ok() ::U
  requires
   [class(U) = operator ∧ isIn(q, m)] ∨
   [class(U) = faculty ∧ sender(m) = id(U)
          ∧ isIn(q, m)]
  ensures q' = del(q, m)

q:: Cancel(m: message) / Unauthorized() ::U
  requires [class(u) = student] ∨
   [class(u) = faculty ∧ sender(m) ≠ id(u)]
  ensures q' = q

q:: Cancel(m: message) / Absent() ::U
  requires [class(U) = operator ∧ ¬isIn(q, m)] ∨
   [class(U) = faculty ∧ ¬isIn(q, m)]
  ensures q' = del(q, m)

q:: List() / Ok(p: queue) ::U
  requires class(U) = operator ∨
           class(U) = faculty
  ensures q = q' ∧
   class(U) = operator ⇒
     ∀ m.(isIn(p', m) ⇔ isIn(q, m)) ∧
   class(U) = faculty ⇒
     ∀ m.(isIn(p', m) ⇔ (isIn(q, m) ∧
                         sender(m) = id(U)))

q:: List() / Unauthorized() ::U
  requires ¬(class(U) = operator ∨
             class(U) = faculty)
  ensures q = q'
```

**Figure 3-2:** Interfaces for Secure Mail Queue

```
q:: Deq() / Ok(m: message) ::U
  requires class(U) = mailer ∨
   class(U) = operator ∨ id(U) = Alice
  ensures q' = del(q, m)

q:: Deq() / Unauthorized() ::U
  requires ¬(class(U) = mailer ∨
   class(U) = operator)
  ensures q' = q
```

**Figure 3-3:** Interfaces if Alice Can Dequeue Messages

```
q:: List() / Ok(p: queue) ::U
   requires class(U) = operator ∨
            class(U) = faculty ∨
            id(U) = Bob
   ensures q = q' ∧
     class(U) = operator ⇒
       ∀ m.(isIn(p', m) ⇔ isIn(q, m)) ∧
     class(U) = faculty ⇒
       ∀ m.(isIn(p', m) ⇔ (isIn(q, m) ∧
                            sender(m) = id(U))) ∧
     id(U) = Bob ⇒
       ∀ m.(isIn(p', m) ⇔ (isIn(q, m) ∧
                            sender(m) = Alice))

q:: List() / Unauthorized() ::U
   requires ¬(class(U) = operator ∨
              class(U) = faculty)
   ensures q = q'
```

**Figure 3-4:** Interfaces if Bob Learns Alice's Password

## 4. Summary

In general, our relaxation lattice method suggests the following design strategy:

- Identify a set of constraints ***C*** that characterizes the preferred behavior. This set induces a lattice $2^{C}$.
- Not all elements in the lattice may correspond to an intuitively meaningful behavior, let alone an acceptable one. The homomorphism $\phi$ determines which elements in the lattice of automata represent acceptable behaviors.
- Given the lattices of constraints and automata, the cost function determines the price one must pay in moving up the lattice of automata toward the preferred behavior.

For the security domain, the objects are the protected resources, the processes that invoke operations on the objects are the people and programs that have access to the resources, and the set of constraints to satisfy is the complement of the set of capabilities of users for accessing the resources.

The relaxation lattice method is a natural way to capture graceful degradation in large, complex, realistic systems. Moreover, the method is quite flexible. Here, we have applied the method to the security domain. We have found it to be an equally appropriate and intuitively appealing method for capturing graceful degradation of replicated objects in fault-tolerant distributed systems, and of highly concurrent objects in transaction-based database systems [10].

## References

[1] D. E. Bell and L. J. LaPadula. *Secure Computer Systems: Unified Exposition and Multics Interpretation.* Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA, March, 1976.

[2] T. Benzel. Analysis of a Kernel Verification. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 125-131. Oakland, California, May, 1984.

[3] K.P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. 10th Symposium on Operating Systems Principles.* December, 1985. Also TR 85-668, Cornell University Computer Science Dept.

[4] J. Chang and N.F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems* 2(3):251-273, August, 1984.

[5] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM* 19(11):624-633, November, 1976.

[6] Gold, Linde, and Cudney. KVM/370 in Retrospect. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 13-23. Oakland, California, May, 1984.

[7] D.I. Good. *Proving Network Security.* Technical Report 125, Institute for Computing Science, University of Texas, Austin, January, 1984.

[8] J. Gray. Notes on database operating systems. *Lecture Notes in Computer Science 60.* Springer-Verlag, Berlin, 1978, pages 393-481.

[9] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software* 2(5):24-36, September, 1985.

[10] M.P. Herlihy and J.M. Wing. Specifying Graceful Degradation in Distributed Systems. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 167-177. August, 1987.

[11] B.W. Lampson. Protection. *ACM Operating Systems Review* 19(5):13-24, December, 1985.

[12] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N Levitt, and L. Robinson. *A Provably Secure Operating System: The System, Its Applications, and Proofs, Second Edition.* Technical Report CSL-116, SRI, May, 1980.