Time in a Multi-Theory Logical Framework

Paolo Mancarella, Alessandra Raffaetà and Franco Turini
Dipartimento di Informatica, Università di Pisa
Corso Italia, 40, I-56125 Pisa, Italy
e-mail: {paolo,raffaeta,turini}@di.unipi.it

Abstract

We present a knowledge representation framework where a collection of logic programs can be combined together by means of meta-level program composition operations. Each object-level program is composed by a collection of extended clauses, equipped with a time interval representing the time period in which they hold. The interaction between program composition operations and time yields a powerful knowledge representation language in which many applications can be naturally developed. The language is given a meta-level semantics which provides also an executable specification. Two applications in the field of business regulations are shown.

1. Introduction

Logic programming has been widely recognized as a powerful knowledge representation tool in various computing domains. It can be used both for procedural and declarative knowledge representation, and consequently it can be used for both programming and program specification, database applications and for knowledge representation and problem solving in artificial intelligence.

The development of logic programming, however, has shown that the basic paradigm is not expressive enough to deal naturally with several computing problems. To overcome some of these limitations, many extensions of logic programming have been studied to improve its knowledge representation and problem solving capabilities, such as the ability of handling negation, constraints, and abstraction mechanisms.

In this paper we propose yet another extension which addresses the handling of temporal information. Even though there are many proposals in the literature in the field of both temporal databases (see, e.g. [16] and references therein) and temporal logic languages (see, e.g. Event Calculus [10, 14], Temporal Prolog [7, 9], Templog [1], Datalog$_{1S}$ [6]), our proposal addresses the handling of time depending knowledge within the multi-theory framework presented in [4, 5]. This framework allows one to represent knowledge as separate logic theories, which can be combined together by means of various meta-level operators. From a deductive database perspective, each logic program (theory) can be viewed as an extended relational database where relations are represented partly intensionally and partly extensionally. The meta-level operators can then be viewed as a means of constructing views by combining multiple databases in various ways.

The need for extending the multi-theory framework with time came out clearly when we addressed knowledge representation problems, especially in the field of business data and procedures, and regulations and laws. A simple example may be useful to clarify the critical points. Consider the problem of representing the fact that a movie ticket is \$5 for kids and \$7 for adults in the first 6 months of 1996 and \$7 for everybody in the rest of the year. We allow one to attach time intervals to a clause, representing the period of time during which the clause is valid. The following theory *BoxOffice* adopts this representation to model the above information about movie tickets.

0-8186-7937-9/97 \$10.00 © 1997 IEEE

BoxOffice:

```
    ticket-cost(5,p) ←
        age(p,a), a ≤ 16 □
        [<Jan 1 1996>,<Jun 30 1996>]

    ticket-cost(7,p) ←
        age(p,a), a > 16 □
        [<Jan 1 1996>,<Jun 30 1996>]

    ticket-cost(7,p) ←
        age(p,_) □
        [<Jul 1 1996>,<Dec 31 1996>]

    age(p,a) ←
        today(d1), born(p,d2), year-diff(d2,d1,a) □
        [<Jan 1 1900>,α]
```

where α stands for a time-point later than any other one, and *year-diff(d2,d1,a)* computes the age a given the current date $d1$ and the date of birth $d2$. The last clause for *ticket-cost* represents the fact that the ticket is $7 in the second part of the year, regardless of the customer's age. Moreover, notice that the theory is *parametric* with respect to the actual day of the year and the customer's birthday. The predicate *today* is defined in a separate theory which represents the current date, e.g.

Today:

```
    today(<May 28 1996> ) ← □
        [<May 28 1996>,<May 28 1996>]
```

and the customer's birthday is given in a separate theory like

Tom:

```
    born(Tom,<May 7 1981> ) ← □
        [<May 7 1981>,α]
```

The previous theories can be combined by means of a union operator ∪ (see Section 2): the query *ticket-cost(s, Tom)* with respect to the combined knowledge

$$BoxOffice \cup Tom \cup Today$$

yields the answer $s = 5$.

As shown in the example, the kinds of applications we are interested in suggest the following decisions about the introduction of time in our framework:

- the primitive notion of time is *an interval of time*
- time intervals are attached to clauses.

Of course, there are many other options, e.g. using time points and relations among them, attaching time information to relations instead of rules and so on. Investigating the impact of these alternative choices on our framework is one of the tasks we intend to pursue in the immediate future.

The paper is organized as follows. Section 2 briefly introduces the operators for combining logic theories. Meta-logic is exploited to provide a formal, and, at the same time, executable semantics to the operators. Section 3 discusses the introduction of time intervals and its semantics, still based on meta-logic. Section 4 deals with the application of the framework discussed so far to representing knowledge in the field of regulations. Section 5 addresses the problem of handling reasoning over joined intervals, that provides the possibility of computing the maximal interval in which an answer holds. Finally, Section 6 outlines our future research plans.

2. Operators for combining logic theories

Program composition operations have been thoroughly investigated in [4, 5], where both their meta-level and their bottom-up semantics are studied and compared. Here, we adopt the meta-level definition of the operations, which is simply obtained by adding new clauses to the well-known vanilla meta-interpreter for logic programs. Stated otherwise, in this view compositions of programs are realized by a meta-interpreter which combines separate programs at the meta-level, without actually building a new program. The reading of the resulting meta-interpreter is straightforward and, most importantly, the meta-logical definition shows that the multi-theory framework can be expressed from inside logic programming itself.

Following Bowen and Kowalski [3], we employ the two-argument predicate *demo* to represent provability. Namely, $demo(x, y)$ represents that the formula y is provable in the object program x.

The *vanilla* meta-interpreter [15] is the simplest application of meta-programming in logic. A general formulation of the vanilla meta-interpreter can be given by means of the *demo* predicate.

$$demo(x, empty) \leftarrow \qquad (1)$$

$$demo(x, (y, z)) \leftarrow demo(x, y),\ demo(x, z) \qquad (2)$$

$$demo(x, y) \leftarrow clause(x, y \leftarrow z),\ demo(x, z) \qquad (3)$$

The unit clause (1) states that the empty goal, represented by the constant symbol *empty*, is solved in any program x. Clause (2) deals with conjunctive goals. It states that a conjunction (y, z) is solved in the program x if y is solved in x and z is solved in x. Finally, clause (3) deals with the case of atomic goal reduction. To solve an atomic goal y, a clause from the program x is chosen and the body of the clause is recursively solved in x.

We adopt the simple naming convention used by Kowalski and Kim in [11]. Object programs are named by constant symbols, denoted by capital calligraphic letters such as $\mathcal{P}$ and $\mathcal{Q}$. Object level expressions are represented by themselves at the meta-level. In particular, object level variables are denoted by meta-level variables, according to the so-called *non-ground representation* [8]. An object level program $\mathcal{P}$ is represented at the meta-level by a set of axioms of the kind $clause(\mathcal{P}, A \leftarrow B) \leftarrow$, one for each object level clause $A \leftarrow B$ in $\mathcal{P}$. For example, we have the following object and meta-level representations, $\mathcal{N}$ and $\mathcal{N}_m$ respectively, of the logic program for natural numbers.

$$\begin{array}{ll} \mathcal{N}: & nat(zero) \leftarrow \\ & nat((s(x)) \leftarrow nat(x) \\ \mathcal{N}_m: & clause(\mathcal{N}, nat(zero) \leftarrow empty) \leftarrow \\ & clause(\mathcal{N}, nat(s(x)) \leftarrow nat(x)) \leftarrow \end{array}$$

Program composition operations can be implemented by meta-logic in a simple and concise way. Each program composition operation is represented at the meta-level by a functor. The meaning of each functor is defined by new clauses added to the vanilla meta-interpreter.

$$clause(x \cup y, z \leftarrow w) \leftarrow clause(x, z \leftarrow w) \qquad (4)$$

$$clause(x \cup y, z \leftarrow w) \leftarrow clause(y, z \leftarrow w) \qquad (5)$$

$$clause(x \cap y, (z \leftarrow u, v)) \leftarrow \qquad (6)$$
$$clause(x, z \leftarrow u),\ clause(y, z \leftarrow v)$$

In the extended framework, the first argument of *clause* represents a composition of programs, referred to as *program expression*, rather than a single program as in the case of the pure vanilla meta-interpreter. The meaning of the clauses (4)—(6) is straightforward. Informally, union and intersection mirror two forms of cooperation among program expressions. Clauses (4) and (5) define the meta-level implementation of the operation $\cup$, where either expression may be used to perform a computation step. For instance, a clause $z \leftarrow w$ belongs to the meta-level representation of $\mathcal{P} \cup \mathcal{Q}$ if it belongs either to the meta-level representation of $\mathcal{P}$ or to the meta-level representation of $\mathcal{Q}$. In the case of intersection, both expressions must agree to perform a computation step. This is obtained in clause (6) exploiting the basic unification mechanism of logic programming and the non-ground representation of object level programs. A program expression $\mathcal{E}$ can be queried by $demo(\mathcal{E}, G)$, where G is an object level goal.

3. Introducing time intervals

In this section we extend the multi-theory framework in order to handle temporal information. We associate time intervals to clauses of object-level programs. The meta-level representation of object-level programs must be extended accordingly as well as the meta-interpreter of the previous section.

It is worth noting that our temporal model can be classified as *historical* in the taxonomy of [13]. Time intervals represent **valid time**, that is the time for which information models reality and corresponds to the actual time for which a relationship holds in the real world. This allows us to query the database at a certain date in the past to obtain the information that held in that period.

We work on discrete time points. Actual time points are represented by elements of the set $\mathbb{N}$ of natural numbers. A special constant α is used to represent a time point later than any others. An interval is a pair $[a, b]$, where $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\alpha\}$. Notice that an interval open to the future is represented by $[a, \alpha]$.

The relation of inclusion between time intervals is denoted by $\sqsubseteq$ and defined as

$$[a, b] \sqsubseteq [c, d] \leftarrow non\_empty([a, b]),$$
$$non\_empty([c, d]), a \geq c, b \leq d$$

where the predicate non_empty states that an interval is not empty and it is defined as

$$non\_empty([a,b]) \leftarrow a \leq b.$$

Moreover, intersection of time intervals is denoted by $\sqcap$ and it is axiomatized as follows:

$$\begin{aligned}
&[a,b] \sqcap [c,d] = [c,b] \leftarrow non\_empty([a,b]),\\
&\qquad non\_empty([c,d]), a \leq c, c \leq b, b \leq d\\
&[a,b] \sqcap [c,d] = [c,d] \leftarrow non\_empty([a,b]),\\
&\qquad non\_empty([c,d]), a \leq c, d < b\\
&[a,b] \sqcap [c,d] = [a,d] \leftarrow non\_empty([a,b]),\\
&\qquad non\_empty([c,d]), c \leq a, a \leq d, d \leq b\\
&[a,b] \sqcap [c,d] = [a,b] \leftarrow non\_empty([a,b]),\\
&\qquad non\_empty([c,d]), c \leq a, b < d.
\end{aligned}$$

Notice that intersection is defined only on *overlapping intervals*. The axiomatization of $<$ and $\leq$ is straigthforward and hence omitted. However it is worth mentioning that $a < \alpha$, for any $a \in \mathbb{N}$.

An object level program is still a collection of clauses named by a constant symbol. Each clause is now equipped with a time interval representing the period of time in which the clause holds. At the object level, an extended clause looks like

$$A \leftarrow B_1, \ldots, B_n \,\Box\, [a,b].$$

According to the above extension, the meta-level interpreter of the previous section must be extended by taking time intervals into account. The predicate *demo* has now an extra-argument denoting a time interval: $demo(\mathcal{E}, G, I)$ means that the goal G holds with respect to the program expression $\mathcal{E}$ and within the time period I. The extended meta-interpreter is defined by the following clauses.

$$\begin{aligned}
&demo(x, empty, I) \leftarrow\\
&demo(x, (y,z), I) \leftarrow demo(x, y, K),\\
&\qquad demo(x, z, J), I \sqsubseteq K \sqcap J\\
&demo(x, y, I) \leftarrow clause(x, y \leftarrow z, K),\\
&\qquad demo(x, z, J), I \sqsubseteq K \sqcap J\\
&clause(x \cup y, z \leftarrow w, I) \leftarrow\\
&\qquad clause(x, z \leftarrow w, K), I \sqsubseteq K\\
&clause(x \cup y, z \leftarrow w, I) \leftarrow\\
&\qquad clause(y, z \leftarrow w, K), I \sqsubseteq K\\
&clause(x \cap y, (z \leftarrow u, v), I) \leftarrow clause(x, z \leftarrow u, K),\\
&\qquad clause(y, z \leftarrow v, J), I \sqsubseteq K \sqcap J
\end{aligned}$$

A clause $x \leftarrow y \,\Box K$ of a plain program P is now represented at the meta-level by

$$clause(P, x \leftarrow y, I) \leftarrow I \sqsubseteq K.$$

In the full version of this paper [12] we have also defined an abstract semantics of the extended framework and proved the correctness of the meta-interpreter defined in this section with respect to that abstract semantics. Such a semantics is obtained by defining a bottom-up operator $\mathcal{F}$ analogous to the immediate consequences operator of standard logic programming. The operator $\mathcal{F}$ is defined over program expressions and an extended Herbrand Base containing pairs of the form (A, I) where A is a ground atom and I is a non-empty time interval. The operator $\mathcal{F}$ is defined as follows:

$$\begin{aligned}
\mathcal{F}_P(\mathcal{I}) = \{(A,I) \mid\ & \exists (B_1, I_1), \ldots, (B_n, I_n) \in \mathcal{I} \,\wedge\\
& A \leftarrow B_1, \ldots, B_n \Box I_0 \in ground(P) \,\wedge\\
& I = \sqcap_{j=0,n} I_j \,\}
\end{aligned}$$

$$\mathcal{F}_{E_1 \cup E_2}(\mathcal{I}) = \mathcal{F}_{E_1}(\mathcal{I}) \cup \mathcal{F}_{E_2}(\mathcal{I})$$

$$\mathcal{F}_{E_1 \cap E_2}(\mathcal{I}) = \mathcal{F}_{E_1}(\mathcal{I}) \cap' \mathcal{F}_{E_2}(\mathcal{I})$$

where:

- $\mathcal{I}_1 \cap' \mathcal{I}_2 = \{(A, I) \mid \exists (A, I_1) \in \mathcal{I}_1, \exists (A, I_2) \in \mathcal{I}_2 : I = I_1 \sqcap I_2\}$
- P is a plain program, E_1 and E_2 are program expressions and $ground(P)$ denotes the set of ground instances of clauses of P.

4. Application to legal reasoning

In this section we show the usefulness of our meta-logic by showing its application to legal-reasoning.

The basic idea is that laws and rules are naturally represented in separate theories and that they can be combined in ways that are necessarily more complex than plain merging. Time is another crucial ingredient in the definition of laws and rules. Quite often, rules have to refer to instant of time and, furthermore, they have a validity for a fixed period of time. This is especially true for laws and rules which concern taxation and government budget related regulations in general.

For the sake of clarity, in the following examples we write object programs as named collections of object clauses (instead of using the clumsier meta-level representation), and we use year dates instead of integer time points.

We present an example concerning a body of Italian regulations dealing with paying taxes on real estate transactions. The original regulation depends on time calculations, since the amount of taxes depend on the period of ownership of the real estate property. Furthermore, the law was abolished in 1992, that means that the rules still apply but only for the period antecedent to 1992.

Our approach allows us to have a theory containing the original regulation and to have two other theories, one containing the constraints due to the decisions taken in 1992, and the other containing the additions. It is important to notice that the design of the constraining theory can be done without taking care of the details (which may be quite complicated) embodied in the original law.

The following theory - INVIM - contains a sketch of the original body of regulations.

INVIM:

```
  due(amount,x,property) ←
        buys(x,property,t1),
        sells(x,property,t2),
        compute(amount,x,property,t1,t2) □
        [<Jan 1 1950>,α]
  compute(amount,x,property,t1,t2) ← . . .
```

In order to adapt the above body of regulations to the new situation imposed by the 1992 decisions, we construct two new theories. The first one is designed as a set of constraints on the applicability of the original rules, while the second one is designed to embody new rules capable of handling the new situation.

CONSTRAINTS:

```
    due(amount,x,property) ←
          sells(x,property,t),
          before(t,<Dec 31 1992>) □
          [<Jan 1 1993>,α]
    compute(amount,x,property,t1,t2) ← □
          [<Jan 1 1993>,α]
```

The first rule specifies that the relation *due* is computed, i.e. its original body is computed, provided that the selling date is antecedent to December, 31 1992. The second rule specifies that the rules for compute, whatever number they are, and whatever complexity they have, carry on unconstrained to the new version of the regulation.

ADDITIONS:

```
  due(amount,x,property) ←
    buys(x,property,t1), sells(x,property,t2),
    after(t2,<Jan 1 1993>),
    compute(amount,x,property,t1,<Dec 31 1992>) □
    [<Jan 1 1993>,α]
```

This rule handles the case of selling the property after the first of January, 1993.

Now, we consider a separate theory representing the transactions regarding a specific individual, say Mary, who bought an apartment on March 8, 1965 and sold it on July 2, 1993.

TRANS:

```
      buys(Mary,Apt8,<Mar 8 1965>) ← □
            [<Mar 8 1965>,α]
      sells(Mary,Apt8,<Jul 2 1993>) ← □
            [<Jul 2 1993>,α]
```

The query

demo(INVIM ∪ TRANS, due(amount,Mary,Apt8),_)

yields the amount, say 118, Mary has to pay when selling the apartment according to the old regulations. On the other hand, the query

$$demo(((INVIM \cap CONSTRAINTS) \cup INVIM - ADDITIONS) \cup TRANS, due(amount, Mary, Apt8), \_)$$

yields the amount, say 102, Mary has to pay when selling the apartment according to the new regulations. Notice the use of the intersection operator as a natural way of imposing constraints on existing theories.

In general, suppose a theory is given that establishes the validity of a certain property, by means of clauses of the form property(x,y,z) $\leftarrow$ Body $\Box$ [a,b].

By intersecting the above theory with a theory containing a clause of the form

property(x,y,z) $\leftarrow$ Body' $\Box$ [c,d].

we constrain the property in two respects. The intersection operator on theories imposes that both *Body* and *Body'* must hold in order to derive the property and that this applies only if the time intervals $[a, b]$ and $[c, d]$ do overlap. Intuitively, this corresponds to a new object level rule

property(x,y,z) $\leftarrow$ Body, Body' $\Box$ I

where $I \sqsubseteq$ [a,b] $\sqcap$ [c,d].

5. Reasoning over joined intervals

The meta-interpreter of Section 3 does not allow us, given a program expression, to compute the maximal interval in which a query holds. For example, consider the following theories representing two library databases:

DB1:
borrow(Mary,The 12th Night) $\leftarrow \Box$
[<May 12 1995>,<Jun 12 1995>]
DB2:
borrow(Mary,The 12th Night) $\leftarrow \Box$
[<Jun 12 1995>, <August 1 1995>]

By querying the union of the above theories we would obtain the period of time in which Mary has borrowed The Twelfth Night.

$demo(DB1 \cup DB2,$ borrow(Mary,The 12th Night)$, i)$

According to the semantics considered so far, the above query computes two answers:

i = [<May 12 1995>,<Jun 12 1995>]
i = [<Jun 12 1995>,<August 1 1995>]

Actually, Mary has borrowed the book from May 12, 1995 until August 1, 1995 which can be obtained by joining the previous answers.

Reasoning over joined intervals requires first to axiomatize the union on intervals, denoted by $\sqcup$.

$$\begin{aligned}
&[a,b] \sqcap [c,d] = [a,d] \leftarrow non\_empty([a,b]),\\
&\qquad non\_empty([c,d]), a \leq c, c \leq b+1, b \leq d\\
&[a,b] \sqcup [c,d] = [a,b] \leftarrow non\_empty([a,b]),\\
&\qquad non\_empty([c,d]), a \leq c, d < b\\
&[a,b] \sqcup [c,d] = [c,b] \leftarrow non\_empty([a,b]),\\
&\qquad non\_empty([c,d]), c \leq a, a \leq d+1, d \leq b\\
&[a,b] \sqcup [c,d] = [c,d] \leftarrow non\_empty([a,b]),\\
&\qquad non\_empty([c,d]), c \leq a, b < d
\end{aligned}$$

It is worth noting that $\sqcup$ is defined on *overlapping intervals* or on *meeting intervals*, that is $[a, b]$ and $[b + 1, c]$.

Then we add the following clause to the meta-interpreter.

$$demo(x, y, I) \leftarrow demo(x, y, K), demo(x, y, J), I = K \sqcup J$$

The above clause allows one to obtain i = [< May 12 1995>,<August 1 1995>] as a computed answer substitution to the query

$demo(DB1 \cup DB2,$ borrow(Mary,The 12th Night)$, i)$.

In [12] the bottom-up semantics is also extended to cope with this form of reasoning.

6. Future work

As mentioned in the introduction, we intend to investigate different representations of time, other than the one based on time intervals attached to program clauses (e.g., using time points and relations among them, attaching time information to relations instead of rules and so on).

In this perspective, we are studying the possibility of replacing time intervals with time points and constraints over them. This would allow us to exploit the technology of constraint logic programming and to get better implementations than the ones provided by meta-interpreters.

Another critical point is the handling of negation. From a practical viewpoint it is sufficient to add the clause

$$demo(exp, not\ x, i) \leftarrow not\ demo(exp,x,i)$$

to the meta-interpreter in order to embody the negation by default of logic programming into our language. However, from a theoretical viewpoint, the interactions between negation by default and program composition operators is still to be fully understood.

Another research direction is about designing more powerful operators. We have two categories in mind:

- Constraint Operators, that allow one to use a program as a set of constraints to apply to other programs. An operator of this kind, operating over programs without time decorations, has been presented in [2].
- Hierarchical operators, i.e. operators that define hierarchical relations among programs.

Acknowledgments: We thank P. Baldan, M. Carboni, S. Contiero and C. Renso for their useful comments and suggestions on earlier versions of this paper. This work has been partially supported by the C.N.R. Project "Environments and tools for the management of temporal information" under grant CTB n. 96.02016.CT12.

References

[1] M. Abadi and Z. Manna. Temporal logic programming. In *Journal of Symbolic Computation*, volume 8, pages 277–295, 1989.

[2] D. Aquilino, P. Asirelli, C. Renso, and F. Turini. Applying restrictions constraints to deductive databases. *Annals of Mathematics and Artificial Intelligence*, 1996. (to appear).

[3] K. A. Bowen and R. A. Kowalski. Amalgamating Language and Metalanguage in Logic programming. In K. L., Tarnlund, and S. A. Clark, editors, *Logic Programming*. Academic Press, 1982.

[4] A. Brogi. *Program Construction in Computational Logic*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1993.

[5] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 1994.

[6] J. Chomicki and T. Imielinski. Temporal Deductive Databases and Infinite Objects. In *Proceeding of ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 61–73, June 1988.

[7] D. M. Gabbay. Modal and temporal logic programming. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 197–237. Academic Press, 1987.

[8] P. M. Hill and J. W. Lloyd. Analysis of Metaprograms. In H. D. Abramson and M. H. Rogers, editors, *Metaprogramming in Logic Programming*, pages 23–52. 1989.

[9] T. Hrycej. Temporal Prolog. In *Proc. of the European Conference on Artificial Intelligence*, pages 296–301, 1988.

[10] R. A. Kowalski and M.J. Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986.

[11] R.A. Kowalski and J.S. Kim. A metalogic programming approach to multi-agent knowledge and belief. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.

[12] P. Mancarella, A. Raffaetà, and F. Turini. Time in a multi-theory logical framework. (In preparation).

[13] R. Snodgrass. Temporal Databases. In *Proceedings of the International Conference on GIS - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, pages 22–64, 1992.

[14] S.M. Sripada. A logical framework for temporal deductive databases. In *Proceedings of the Very Large Databases Conference*, pages 171–182, 1988.

[15] L. Sterling and E. Shapiro. *The Art of Prolog*. 1986.

[16] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and editors R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.