# RTSORAC: A Real-Time Object-Oriented Database Model [⋆]

J. J. Prichard, Lisa Cingiser DiPippo, Joan Peckham, Victor Fay Wolfe

Department of Computer Science and Statistics
The University of Rhode Island, Kingston, RI, 02881 USA
*lastname@cs.uri.edu*

**Abstract.** A *real-time database* is a database in which both the data and the operations upon the data may have timing constraints. We have integrated real-time, object-oriented, semantic and active database approaches to develop a formal model called RTSORAC for real-time databases. This paper describes the components of the RTSORAC model including objects, relationships, constraints, updates, and transactions.

## 1 Introduction

A *real-time database* is a database in which both the data and the transactions may have timing constraints. Typically, a real-time database manages data from the environment, processes environmental information in the context of previously acquired information and provides a timely response to transactions that use the data [7]. Much of the current research has been directed towards developing relational real-time databases [7, 8]. Although the relational model is useful for many applications, we believe that it is not as well-suited as an *object-oriented database model* (OODM) for applications that require complex data, complex relationships among data, first-class support for timing constraints, and more scheduling flexibility than serializability can provide. RTSORAC (**R**eal-**T**ime **S**emantic **O**bjects **R**elationships **A**nd **C**onstraints) is a real-time object-oriented database model that incorporates these concepts.

The RTSORAC model is based upon an earlier model called SORAC [5]. The SORAC data model combines features of the object-oriented [13] and semantic data models [6]. The prototype implementation of the SORAC model translates object and relationship definitions into code that executes on a commercial object-oriented database system. In SORAC, enforcement rules specifying the explicit means for maintaining interobject constraints are directly specifiable by the database designer.

This paper represents an extension of the SORAC model for real-time applications. We have drawn from our experience in the design and implementation of the *RTC* real-time programming language constructs [11] and SORAC to identify the basic research issues involved in the design of real-time object-oriented

[⋆] This material is based upon work supported by the U.S. National Science Foundation under grant No. IRI-9308517 and the U.S. Naval Undersea Warfare Center

databases (RTOODBs). The next section of the paper provides background information on real-time databases, and summarizes the properties that are unique to real-time databases. The third section describes the RTSORAC model, and the last section describes the ongoing implementation of the RTSORAC model. Issues for future investigation are also briefly mentioned.

## 2 Real-Time Databases

A real-time database system has two distinguishing features: the notion of temporally consistent data, and the ability to place real-time constraints on transactions. These features are useful to time critical applications that need to collect, modify, and retrieve shared data. Since the data used by these applications must closely reflect the current state of the application environment, we need mechanisms for measuring this closeness. These measures are based upon time intervals that specify the temporal consistency of the data. Temporal consistency can be measured in two ways: absolute consistency and relative consistency [7]. A piece of data is considered absolutely consistent with respect to time as long as the age of a data value is within a given interval. For example, in a radar system, the data corresponding to a contact, such as its speed, should be updated often, (e.g. every five seconds). Hence, the value of the speed is temporally consistent as long as it is no more than five seconds old. Relative consistency is of interest when multiple values are used in computations. It provides a mechanism for checking the ages of the multiple data values with respect to each other. For example, if the radar system computes the new location of a contact using the speed and bearing, it would be important that the ages of the speed and bearing be relatively close to one another (e.g. within two seconds).

Timing constraints on transactions come from one of two sources. First, temporal consistency requirements of the data impose timing constraints on a transaction. For instance, the period of a sensor transaction is dictated by the valid time of the sensor data that it writes. The second source of timing constraints on transactions is system or user requirements on reaction time. There are typically two types timing constraints on transactions: absolute timing constraints (i.e. earliest start time, latest finish time) and periodic timing constraints. Given the added dimension of time on transactions, one of the interesting areas of study in real-time databases is that of transaction scheduling [2, 12]. Not only must the schedules meet timing constraints, they must also maintain the logical consistency of the data in the database. An additional challenge is to provide a strategy for recovery that adheres to the temporal and logical consistency requirements of the system.

Hence, a real-time database system should provide support for specifying:

- the absolute validity interval of a data value
- relative temporal consistency among a set of data values
- absolute timing constraints on transactions
- periodic timing constraints on transactions
- recovery from violations of timing constraints

$$\begin{aligned}
\textbf{Object} &= \langle N, A, M, C, CF \rangle \\
N &= UniqueID \\
A &= \{a_1, a_2, ..., a_m\} \text{ where attribute } a_i = \langle N_a, V, T, I \rangle \\
M &= \{m_1, m_2, ..., m_n\} \text{ where method } m_i = \langle N_m, Arg, Exc, Op, OC \rangle \\
C &= \{c_1, c_2, ..., c_s\} \text{ where constraint } c_i = \langle N_c, AttrSet, Pred, ER \rangle \\
CF &= \text{compatibility function}
\end{aligned}$$

**Fig. 1.** Object characteristics in RTSORAC

## 3 The RTSORAC Model

RTSORAC has three components that model the properties of a real-time object-oriented database: *objects*, *relationships* and *transactions*. *Objects* represent database entities. *Relationships* represent associations among the database objects. *Transactions* are executable entities that access the objects and relationships in the database.

### 3.1 Objects

An *object* (Figure 1) consists of five components, $\langle N, A, M, C, CF \rangle$, where $N$ is a unique name or identifier, $A$ is a set of attributes, $M$ is a set of methods, $C$ is a set of constraints, and $CF$ is a compatibility function. Attributes, methods, constraints, and the compatibility function are described below. Figure 2 illustrates an example of a **Train** object (adapted from [1]) for storing information about a railroad engine in a database.

**Attributes.** $A$ is set of attributes for the object, where each attribute is characterized by $\langle N_a, V, T, I \rangle$. $N_a$ is the name of the attribute. The second field, $V$, is used to store the value of the attribute, and may be of some complex data type. The next field, $T$ is used to store the timestamp of the attribute, and is of some data type capable of expressing a time. Access to the timestamp of an attribute is necessary for determining temporal consistency of the attribute. For example, in the **Train** object, there is an attribute for storing the oil pressure called `OilPressure` to which a sensor regularly provides readings. This update is expected every thirty seconds, thus the `OilPressure` attribute is considered temporally inconsistent if the update does not occur within that time frame. The timestamp value of the `OilPressure` attribute must be utilized by the system to determine that the update did not occur as expected.

The last field $I$ is used to store the amount of imprecision associated with the attribute, and is of the same type as the value field $V$. In order to meet real-time constraints it may not be possible to maintain precise data values. Furthermore, many real-time control applications allow a certain amount of imprecision. For instance, within the **Train** object, the value of `OilPressure` attribute may not have to be precise.

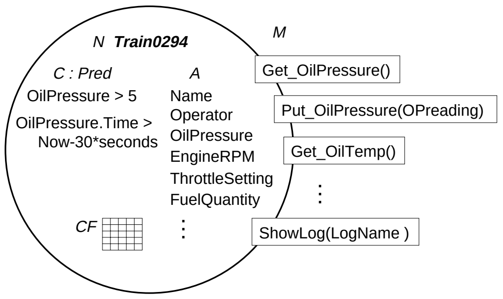

**Fig. 2.** Example of **Train** object

**Methods.** The third component of an object, $M$, is a set of methods, where each method is of the form $\langle N_m, Arg, Exc, Op, OC \rangle$. $N_m$ is the name of the method. $Arg$ is a set of arguments for the method, where each argument has the same components as an attribute, and is used to pass information in and/or out of the method. $Exc$ is a set of exceptions that may be raised by the method to signal that the method has terminated abnormally. $Op$ is a set of operations which represent the implementation of the method. These operations include statements for conditional branching, looping, I/O, and reads and writes to an attribute's value, time, and imprecision fields.

The last characteristic of a method, $OC$, is a set of operation constraints. An operation constraint is of the form $\langle N_{oc}, OpSet, Pred, ER \rangle$ where $N_{oc}$ is the name of the operation constraint, $OpSet$ is a subset of the operations in $Op$, $Pred$ is a Boolean expression, and $ER$ is an enforcement rule. The predicate is specified over $OpSet$ to express precedence constraints, execution constraints, and timing constraints [11]. The enforcement rule is used to express the action to take if the predicate evaluates to false. A more complete description of an enforcement rule can be found in the next section on constraints.

Here is an example of an operation constraint predicate in the **Train** object:

$Pred$ : `complete(Put_OilPressure) < NOW + 5*seconds`

A deadline of `NOW + 5*seconds` has been specified for the completion of the `Put_OilPressure` method. Note the use of a special atom `complete(e)`, which represents the completion time of the executable entity `e`. Other atoms that are useful in the expression of timing constraints include `start(e)`, `wcet(e)`, and `request(e)` which represent the execution start time, worst case execution time, and the execution request time of entity `e` respectively.

**Constraints.** The fourth component of an object is a set of constraints, $C$, which permits the the specification of correct object state. Each constraint is of the form $\langle N_c, AttrSet, Pred, ER \rangle$. $N_c$ is the name of the constraint. $AttrSet$ is a subset of attributes of the object. $Pred$ is a Boolean expression that is specified using attributes from the $AttrSet$. The predicate can be used to express the logical and temporal consistency requirements of the data stored in the object by referring to the value, time, and imprecision fields of the attributes in the set.

The enforcement rule ($ER$) is executed when the predicate evaluates to false, and is of the form $\langle Exc, Op, OC \rangle$. As with methods, $Exc$ is a set of exceptions that the enforcement rule may signal, $Op$ is a set of operations that represent the implementation of the enforcement rule, and $OC$ is a set of operation constraints on the execution of the enforcement rule.

Logical and temporal consistency constraints on data require two distinct methodologies for evaluation. Predicates based upon logical consistency requirements are evaluated when write operations are performed on the attributes in $AttrSet$. All writes in the database are the result of a transaction which may be either user initiated or system initiated. Hence an enforcement rule associated with such a predicate will always be executed in the context of a transaction. This execution may be synchronous or asynchronous and may involve signaling an exception that is propagated back to the transaction. Predicates based upon temporal consistency requirements may be violated simply due to the passage of time and the semantics of predicate evaluation can vary. Once a constraint violation has been detected, the corresponding enforcement rule is executed. It is possible that there is no context (such as a transaction) for the execution of the enforcement rule. In this case the implementation must provide a means of handling exceptions raised outside of the context of a transaction, perhaps through the use of a monitor that can detect and act upon signaled exceptions.

For example, as mentioned earlier, the **Train** object has an oil pressure attribute that is updated with the latest sensor reading every thirty seconds. To maintain the temporal consistency of this attribute, the following constraint is defined:

```
N :        OilPressure_avi
AttrSet :  {OilPressure}
Pred :     OilPressure.time <= Now - 30*seconds
ER :       if Missed <= 2 then
             OilPressure.time = Now
             Missed = Missed + 1
             signal OilPressure_Warning
           else signal OilPressure_Alert
```

The enforcement rule specifies that if only one or two of the readings have been missed, a counter is incremented indicating that a reading has been missed and a warning is signaled using the exception `OilPressure_Warning`. If more than two readings have been missed, then an exception `OilPressure_Alert` is signaled, which might lead to a message being sent to the train operator. The counter `Missed` is reset to zero whenever a new sensor reading is written to attribute `OilPressure`.

**Compatibility Function.** The last component of an object, $CF$, is a compatibility function that expresses the semantics of simultaneous execution of each ordered pair of methods in the object. For each ordered pair of methods, $(m_i, m_j)$,

$$\begin{aligned}
&\textbf{Relationship} = \langle N, A, M, C, CF, P, IC \rangle\\
&\quad N \quad = UniqueID\\
&\quad A \quad = \{a_1, a_2, ..., a_m\} \text{ where attribute } a_i = \langle N_a, V, T, I \rangle\\
&\quad M \quad = \{m_1, m_2, ..., m_n\} \text{ where method } m_i = \langle N_m, Arg, Exc, Op, OC \rangle\\
&\quad C \quad = \{c_1, c_2, ..., c_r\} \text{ where constraint } c_i = \langle N_c, AttrSet, Pred, ER \rangle\\
&\quad CF \quad = \text{compatibility function}\\
&\quad P \quad = \{p_1, p_2, ..., p_s\} \text{ where participant } p_i = \langle N_p, OT, Card \rangle\\
&\quad IC \quad = \{ic_1, ic_2, ..., ic_t\} \text{ where interobject constraint}\\
&\qquad\qquad ic_i = \langle N_{ic}, PartSet, Pred, ER \rangle
\end{aligned}$$

**Fig. 3.** Relationship characteristics in RTSORAC

a Boolean expression ($BE_{i,j}$) is defined. $BE_{i,j}$ is evaluated to determine whether or not $m_i$ and $m_j$ can execute concurrently. In many object-oriented systems, the execution of a single method of an object prevents any other methods of the object from being executed, i.e. the entire object is locked upon invocation of a single method. Through the use of the compatibility function, the designer of an object can allow more flexibility by defining the semantics of the compatibility of each pair of methods. By allowing a higher degree of concurrent access to the object through its methods, perhaps even relaxing serializability, the affected data may become imprecise. An in depth discussion of the semantic locking technique that utilizes the compatibility function to provide concurrency control to an object in RTSORAC can be found in [4].

Consider the following examples of compatibility function specifications:

```
CF(Get_OilPressure(), Get_OilTemp()) = TRUE
CF(Put_OilPressure(OP_reading), ShowLog(Log)) = (Log <> "OilPressure")
```

In the first example, the compatibility function is used to specify that the methods `Get_OilPressure` and `Get_OilTemp` of the **Train** object can always run concurrently (always `TRUE`). This is appropriate since these two methods operate on different attributes, `OilPressure` and `OilTemp`. The second example specifies that `Put_OilPressure` and `ShowLog` can run concurrently as long as the log to be displayed is not "OilPressure". If the requested log is "OilPressure", then the execution of the `ShowLog` method may be delayed or aborted.

## 3.2 Relationships

Relationships represent aggregations of two or more objects. In the RTSORAC model, a *relationship* (Figure 3) consists of $\langle N, A, M, C, CF, P, IC \rangle$. The first five components of a relationship are identical to the same components in the definition of an object. In addition, objects that can participate in the relationship are specified in the participant set $P$, and a set of interobject constraints is specified in $IC$.

Figure 4 illustrates an example of a **Energy Management** relationship for relating a **Train** object with a **Track** object. The **Track** object stores information such as track profile and grade, speed limits, maximum load, and power

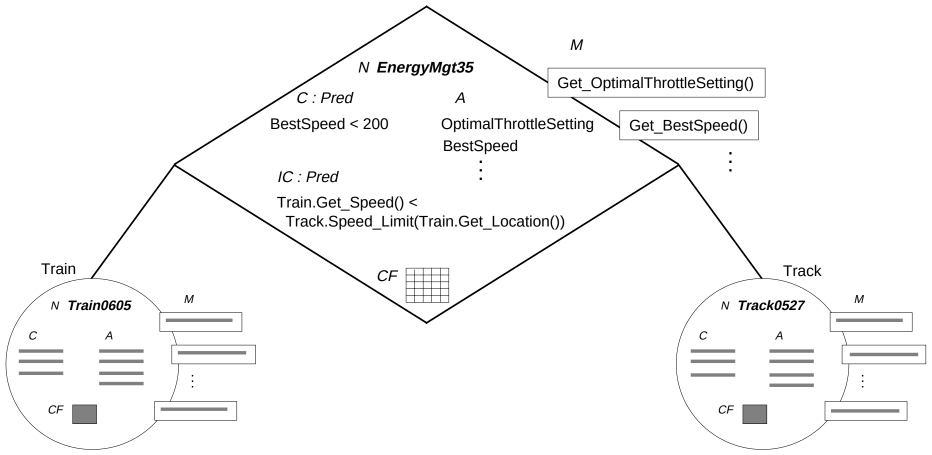

**Fig. 4.** Example of **Energy Management** relationship

available. The energy management relationship uses both train and track information to determine fuel efficient throttle and brake settings.

**Participants.** $P$ is a set of participants in the relationship, each participant is of the form $\langle N_p, OT, Card\rangle$. $N_p$ is the name of the participant. $OT$ is the type of the object participating in the relationship. $Card$ is the cardinality of the participant, which is either *single* or *multi* [3]. Constraints can be used to express cardinality requirements of the relationship, such as minimum and maximum cardinality of the participants. In Figure 4, `Train` and `Track` are single cardinality participants.

**Interobject Constraints.** $IC$ is a set of interobject constraints placed on objects in the participant set, and is of the form $\langle N_{ic}, PartSet, Pred, ER\rangle$. $N_{ic}$, $Pred$, and $ER$ are as in object constraints, and $PartSet$ is a subset of the relationship's participant set $P$. The predicate is expressed using objects from the $PartSet$, allowing the constraint to be specified over multiple objects participating in the relationship. Enforcement rules are defined as before by $\langle Exc, Op, OC\rangle$, however the operations $Op$ can now include invocations of methods of the objects participating in the relationship.

As an example of an interobject constraint, consider the **Energy Management** relationship in Figure 4. A **Train** object will be on one specific segment of track, represented by the **Track** object participating in the relationship. The train should obey the speed limits set on the track, so the following interobject constraint predicate could be specified:

$Pred:$ `Train.Get_Speed()` $<$ `Track.Speed_Limit(Train.Get_Location())`

If the speed of the train exceeds the speed limit posted at the train's location on the track, then the corresponding enforcement rule signals `SpeedLimitExceeded`.

## 3.3 Transactions

A *transaction* has six components, $\langle N_t, O, OC, PreCond, PostCond, Result\rangle$, where $N_t$ is a unique name or identifier, $O$ is a set of operations, $OC$ is a set of operation constraints, $PreCond$ is a precondition, $PostCond$ is a postcondition, and $Result$ is the result of the transaction. Each of these components is briefly described below.

**Operations.** $O$ is set of operations that represent the implementation of the transaction. These operations may include method invocations ($MI$), initiations of subtransactions, *commit* or *abort* statements, and statements for conditional branching, looping, and reads/writes on local variables. A subtransaction initiation allows for transactions to appear within the scope of other transactions. Method invocations ($MI$) are of the form $\langle MN, ArgInfo\rangle$, where $MN$ is the method name (prepended with the appropriate object id) and $ArgInfo$ is a set of tuples containing argument information. Each tuple is of the form $\langle aa, maximp, tcr\rangle$ where $aa$ is the actual argument to the method, $maximp$ is the maximum allowable imprecision of the argument, and $tcr$ is the temporal consistency requirement of the argument. The fields $maximp$ and $tcr$ are specified only for arguments that are used to return information to the transaction. These fields allow the transaction to specify requirements that differ from those defined on the data in the objects. For example, the transaction might be willing to accept a value whose temporal consistency requirements have been violated so as to meet other timing constraints. The data may still be useful to the transaction because of other available information (for example, it may be able to do some extrapolation). A transaction may also specify that data returned by a method invocation must be precise ($maximp$ is zero).

**Operation Constraints.** $OC$ is a set of constraints on operations of the transaction. These constraints are of the same form as the operation constraints specified for methods, $\langle N_c, OpSet, Pred, ER\rangle$. As with methods, these constraints can be used to express precedence constraints, execution constraints, and timing constraints. For example, a transaction may require that a sensor reading which has been stored in the database be returned within two seconds.

**Precondition, Postcondition, Result.** $PreCond$ represents preconditions that must be satisfied before a transaction is made ready for execution. For example, it may be appropriate for a transaction to execute only if some specified event has occurred. The event may be the successful termination of another transaction, or a given clock time. $PostCond$ represents postconditions that must be satisfied upon completion of the operations of the transaction. The postconditions can be used to specify the semantics of what constitutes a *commit* of a transaction containing subtransactions. $Result$ represents information that is returned by the transaction. This may include values read from objects as well as values computed by the transaction.

$$\begin{tikzcd}
\text{SORAC model} \\[-20pt]
\textit{OIL} \arrow[d] & & \textit{RTOIL} \arrow[d] \\
\textit{C++ objects} \arrow[d, shift left] \arrow[d, shift right] & & \textit{C++ objects} \arrow[d, shift left] \arrow[d, shift right] \\
\texttt{ONTOS DB system} & & \texttt{Real-time OpenOODB system}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{RTSORAC model}
\end{tikzcd}$$

**Fig. 5.** Mapping of the models to database systems

## 4 Implementation

As mentioned in the introduction of this paper, RTSORAC is based upon an earlier model called SORAC. Figure 5 illustrates the prototype implementation of the SORAC model, and shows the parallel implementation of RTSORAC. The implementation of SORAC provides a data definition language called OIL (Object Interaction Language), which allows the specification of objects and relationships, as well as interobject constraints [5]. These specifications are compiled into standard C++ object definitions with calls to the underlying ONTOS object-oriented database system.

A similar approach is proposed for implementation of RTSORAC. There are two main efforts involved with this implementation. First is the extension of Open OODB [9] to support objects and transactions that have real-time characteristics. Thus, the real-time version of Open OODB will replace ONTOS as the underlying database system. Second is the real-time extension of OIL (RTOIL) to support characteristics of the RTSORAC model that do not appear in SORAC. These characteristics include the compatibility function for concurrency control, extended data types to support time, and the incorporation of time into the specification of constraints upon objects and relationships. RTOIL must automate the mapping of each component of an object or relationship to the attributes and methods of standard C++ objects with library calls to real-time version of Open OODB.

The effort to extend Open OODB for real-time will involve modification or replacement of many of the components of Open OODB. As an alpha site for Open OODB, we have had an opportunity to gain a full understanding of its modular implementation structure. Open OODB is organized as a series of replaceable policy managers such as the Transaction Policy Manager, and the Persistence Policy Manager. We have recently implemented a prototype Object Policy Manager capable of supporting a RTSORAC object. Details on the design of our real-time extensions to Open OODB can be found in [10].

## 5 Conclusions

This paper has provided a general model for real-time object-oriented databases. The model combines features of object-oriented databases, semantic data models, real-time databases, and active databases. Our current implementation experiences using Open OODB indicate the applicability of the RTSORAC model.

As the design and implementation of the RTSORAC model progresses, a number of issues related to the RTSORAC model will be investigated. These have been identified and include inheritance, query language support, recovery techniques, and necessary operating system support. While these represent a diverse set of database issues, we have discovered that none can be investigated in isolation. Thus, of particular interest will be the unique interaction among these constructs that arise in the real-time environment.

**Acknowledgements.** We thank Paul Fortier, Gary Hyslop, and John Black for their helpful comments and suggestions.

## References

1. Grady Booch. *Object-Oriented Design.* The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.
2. A.P. Buchmann, D.R. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *The Fifth International Conference on Data Engineering*, February 1989.
3. Oscar Diaz and Peter M.D. Gray. Semantic-rich user-defined relationship as a main constructor in object-oriented databases. In R.A. Meersman, W. Dent, and S. Khosla, editors, *Object-Oriented Databases: Analysis,Design & Construction (DS4).* Elsevier Science Publishers, B.V. (North-Holland), 1991.
4. L. Cingiser DiPippo and V. Fay Wolfe. Object-based semantic real-time concurrency control. *Proceedings of the 14th Real-time Systems Symposium*, Dec. 1993.
5. Michael Doherty, Joan Peckham, and Victor Fay Wolfe. Implementing relationships and constraints in an object-oriented database using monitors. In *Rules in Database Systems, Proceedings of the 1st International Workshop on Rules in Database Systems.* Springer-Verlag, August 1993.
6. Joan Peckham and Fred Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3):153–189, September 1988.
7. Krithi Ramamritham. Real-time databases. *International Journal of Distributed and Paralled Databases*, 1(2), 1993.
8. Sang Son, editor. *ACM SIGMOD Record (Special Issue on Real-Time Database Systems).* ACM Press, March 1988.
9. David Wells, Jose Blakely, and Craig Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10), Oct. 1992.
10. V. F. Wolfe, L. C. DiPippo, J. J. Prichard, J. M. Peckham, and P. J. Fortier. The design of real-time extensions to the open object-oriented database system. Technical Report URI-TR94-236, University of Rhode Island, Department of Computer Science, May 1994.
11. Victor Wolfe, Susan Davidson, and Insup Lee. *RTC*: Language support for real-time concurrency. *Real-Time Systems*, 5(1):63–87, March 1993.
12. P. Yu, K. Wu, K. Lin, and S. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82, January 1994.
13. Stanley Zdonik and David Maier. *Readings in Object Oriented Database Systems.* Morgan Kauffman, San Mateo, CA, 1990.