Design of Real time Systems Using Object-Oriented Model

Ramesh Krishnan

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada, H3G-1M8

December 1994

ISBN 0-612-01381-2

Canada

Abstract

Design of Real-time Systems Using OO Model

Ramesh Krishnan

The proper design of real-time systems has been an active area of research for a long period of time. Real-time systems acquire input from multiple devices at an extremely fast rate. These systems frequently require concurrent processing of multiple inputs. The predictability and reliability expected from these systems is extremely high. From a software engineering perspective, the design of these systems should permit dynamic changes to any of the system parameters thereby making it flexible. In addition, the need for reusable software modules to reduce development costs for future applications is a requirement being imposed by the software development community.

With the emergence of new software engineering techniques such as the Object Oriented (OO) paradigm, the utilization of these methodologies for the real-time domain is being investigated. The OO paradigm has been shown to be an improvement over the function-oriented approach, in the areas of understandability, adaptability, and code reusability. Several projects are underway to use this approach towards the design of flexible, and predictable real time systems.

An analysis on using the OO model for the design of real-time systems has been presented in this report. The main characteristics of a real-time object-oriented model have been outlined. In addition, this area has been further explored through the implementation of a Railroad crossing system using a real-time object-oriented model known as **TROM(Timed-Reactive Object Model)**. The Railroad crossing application demonstrates all the essential requirements of a real-time system. This report concludes with the specification, design, and implementation issues of this application.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Introduction

Real time systems are a special breed of computer applications that have been studied by several researchers over a long period of time. These computer systems demonstrate certain specific characteristics which need to be addressed during their specification, design and implementation phases. For example, an automobile computer that generates timing must produce a spark at a certain precise instant, even if there are other tasks that the computer must carry out. As the number of critically time dependent tasks in a computer system grows, the computer programs that execute and monitor these real-time constraints must be carefully designed. The most important component is the development of the underlying models used to represent the systems. Such a model should ideally possess formal semantics that allow the system's correctness to be verified, at the same time enabling the design to represent the software and real world entities in a way that feels natural to system designers. Several approaches have been suggested to address the special needs of these systems. One such approach is the Object Oriented (OO) Software Engineering Model. This model has shown to be an improvement over traditional procedural programming, particularly in the areas of understandability, adaptability, and code reusability. However, in the area of real-time systems, the OO model has caught on more slowly.[10] This report contains an introduction to the area of real-time systems and examines how the OO model can be effectively used to represent the temporal characteristics of real-time systems. In addition, an example of a real-time application is studied and an object-oriented design of the system using the **TROM^2 (Timed-Reactive Object Model)** is presented.

Real-time Systems

Whenever a computer is required to acquire data, emit data, or interact with its environment at precise times, the system is said to be a real-time computer system. There are several applications that demonstrate these common set of requirements. For example, it is common to refer to systems that operate in an on-line, interactive environment and that require fast response times as "real-time systems". Let us consider the requirement of an on-line banking system, from the point of view of a user : enter, record and report transactions that occurred on an account or between accounts in an accurate manner. This will also be accompanied by a quantitative constraint, such as "report the result of the transaction within two seconds". The "real-time" designation in a system of this type arises because of the specified quantitative constraint which

is a response time requirement of the underlying system software on top of which the banking application runs Therefore, by shifting the boundary from the system software to the application, we can describe the system in banking terms from the point of view of a bank clerk, bank manager, or an account user, who wants to know nothing about the real-time issues. On the other hand, there is another set of applications whose specification entails constraints related to time in the real world. Applications such as aircraft control system, digital exchange system, factory robotics, air-traffic control system, etc. fall into this category. The following is a list of characteristics of these systems that distinguish them from other software systems :

1. *The environment of a real-time system often contains devices that provide input to the system.* In a broad sense any system that accepts input may be said to be sensing what is occurring in its environment. However, non-real-time systems are restricted to inputs that occur at discrete points in time and are highly structured (for example, keyboard or data line inputs). A real-time system, on the other hand, is typically attached to sensors and thermocouples, optical scanners, and contact probes, and can thus collect a continuous stream of relatively unstructured data.

2. *Real-time systems often require concurrent processing of multiple inputs.* Nearly all non-trivial systems require inputs from multiple sources. However, a real-time application, such as an industrial process control system, might be required to correlate values of temperature, pressure, and perform simultaneous adjustments of values in the loop to maintain a process in a desired state. On the other hand, a system handling on-line inquiries from two users that are entered at about the same time, need not be truly concurrent although it may be implemented that way, but must merely respond to both transactions within a short time interval

   The concurrent processing requirement is satisfied by incorporating concurrent tasks in software design.[3] This model reflects the natural parallelism that exists for many real-world application, than the traditional sequential program model.

3. *The time-scales of many real-time systems are fast by human standards.* In terms of exchange of information between human beings or between human beings and automated systems, one second is not a long delay. On the other hand, the devices that real-time systems monitor and control, often operate on time scales in which a second is an extremely long time. An automobile cruise control system, adjusts the throttle based on measurements of current speed to insure that the desired speed is maintained. The actual speed must be monitored many times per second in order to maintain a smooth

ride. Although this is rapid by human standards, it is on the low end of the spectrum in terms of real-time requirements.

4. *The precision of response required of real-time systems is greater than that required of other systems.* Real-time systems have timing constraints, that is, they process events within a given time frame. In an interactive system, a human may be inconvenienced if the system response is delayed, whereas, in a real-time system a delay may be catastrophic. For example, inadequate response in an air traffic control system could result in a mid-air collision of two aircraft The required response time will vary by application.

   Based on the timeliness requirements there are two major categories of industrial real-time systems, namely **hard and soft[4] systems**. Hard real-time systems require that their timing requirements be met, otherwise a catastrophe can occur, and soft systems are those where services are provided in real-time and while important, a catastrophe will not occur if immediate service is not provided. The control of a modern aircraft belongs to the hard category; whereas a digital telephone exchange is a typical example of the non-hard category.

The list of characteristics given does not constitute a definition; a system need not have all these characteristics to be considered a real-time system. A given system may be clearly in the real-time category and be lacking requirements for concurrence. In addition, the discussions above does not imply the absence of real-time problems in the banking or database environment Real-time problems do exist in these applications and are best addressed, however, by separating the systems completely from the applications they will serve when completed.

Real-time Software design issues

The core of a real-time system consists of a task with a specified timeliness requirement. Such task is known as a *real-time task.* Each task needs to complete its operations within a specified time. The requirements of a task needs to be captured and implemented in an efficient manner. Therefore, the design of the system should enable the specification, and verification of a task's time-constraints such as the start-time, deadlines, and periods. However, as the number of time-constrained tasks increases, the number of events that occur in parallel increases, thereby making it difficult to specify, design and verify such systems.

The fundamental challenge in the design of real-time systems is how to incorporate the time parameter. Timing constraints for tasks can be arbitrarily complicated, but they can be classified into two broad classes **Periodic** tasks and **Aperiodic** (or sporadic) tasks.[7] A periodic task has a deadline by which it must finish or start, or it may have a constraint on both start and finish times. Aperiodic task's requirements can arise from dynamic events such as an object falling in front of a moving robot or a human operator pushing a button on a console. Therefore, constructs in the system must support the specification and management of time for both classes of tasks. In addition, facilities must be provided to raise an exception when a task's deadline is missed.

In addition to timing constraints, a task may also possess the following types of constraints:

- Resource requirements: A task may require access to certain resources other than the CPU such as I/O devices, data structures, files and databases. The requests for access to these resources arrive randomly, and the system must deal with these unpredictable events and guarantee their satisfaction.
- Severity: Depending on the functionality of a task, meeting the deadline of one task may be considered more critical than another. For example, a task that reacts to an emergency situation such as fire on the factory floor, in all likelihood, will be more critical than the task that controls the movement of a robot under normal operating conditions. The same notion of task priority or severity needs to exist in the system.

Furthermore, from a Software design perspective, the design of reusable real-time software modules plays an important role. The investigation of reusable software modules is an important subject, because it reduces the software development cost and enhances the quality of the resulting software. Reusable real-time software modules have the added difficulty of meeting different timing requirements for different applications.

Software Systems Development Paradigm

Problem solving is the fundamental activity of people working in science and engineering. It has two aspects: *process and structure*.[9] The process begins with a problem statement (set of requirements) and ends with a solution statement (set of specifications). The *process* is generally called reasoning and the result of reasoning is structure. The need for structure arises from the need to understand the result of reasoning.

There are two ways of representing structure: *function* and *form*.[9] *Function* can be represented by codifying behavior, and *form* can be represented by classifying features. So one result of reasoning or thinking about a problem is a set of "rules" of behavior, and another result is a set of "classes" of features.

A way of thinking about something is sometimes called a *paradigm*. Hence, the result of reasoning about *function* is called the **rule-based paradigm,**[9] and the result of reasoning about *form* is called the **class-based paradigm.** Since classes are composed of objects, the class-based paradigm is also known as **object-oriented paradigm**. Traditionally, science seems more interested in form and engineering more interested in function. However, both disciplines are interested in what happens to their structures, whether form or function.

There are two basic system development paradigm: **Function/data** oriented paradigm or **object-oriented** paradigm. The **Function/data** paradigm views the function of a system and the data required by the system as two separate entities. The object-oriented paradigm views function and data as highly integrated. These paradigms are shown schematically in Figure 1.

The Function/data paradigm distinguishes between functions and data, where functions, in principle, are active and have behavior, and data are passive holders of information which become affected by functions. The system is typically broken down into functions, and data is sent between those functions. The differences between the two paradigms is listed in Table 1.

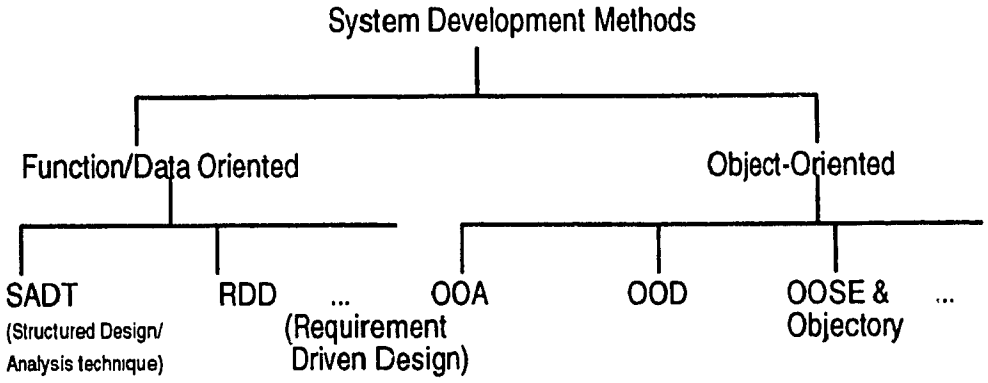


Figure 1 : Different Software Development Paradigms

OO Design	Function-Oriented Design
Problem decomposition yields software objects that encapsulate data and functions.	Problem decomposition results in a collection of modules, each responsible for a single step in the overall process.
Data is encapsulated within the object	Data is distributed globally throughout the system
Close mapping between software objects and "real" or "physical" objects in the application environment.	Loose mapping between software modules and the "objects" of the real world.
The close mapping allows internal changes to a "real" object and its corresponding software object to be affected without affecting the function of other objects in the system.	Due to the loose mapping, a change in the application could require modification to many modules. This could make maintainability more difficult, and also prevents reuse of the software modules.
OOD object structuring criterion are based on information hiding and do not take into account concurrency and timing issues.	A few function-oriented methods, **DARTS and JSD,** place considerable emphasis on concurrent task structuring.
Finite state machines are not given much importance. Timing diagrams are used to show timing dependencies between objects.	Finite state machines specify the control aspects of a system. The timing constraints are specified during the specification and design stage through the use of network diagrams.

Table 1 : Object-Oriented Design Vs. Function-Oriented Design

Basic concepts of Object-Oriented Model

The OO model is made up of two types of entities: Objects and messages. Both physical entities and software entities are represented as objects.

Software objects encapsulate data and provide units of code called *methods* for accessing the data. Each object has a unique identity, which other objects use to address it. Objects communicate by sending and receiving object and contain the name of a specific method of the receiving object that is to be executed along with parameters a such as the names of other objects. Sending a message to an object with the name of a specific method is the only mechanism by which an object can manipulate another object's encapsulated data. The object oriented model thus provides **data and program abstraction,** hiding details of an object's implementation from other objects.

A class represents all objects with same set of characteristics; classes can have subclasses. A subclass represents all objects that belong to the same (super) class, and that possess some additional common characteristics. In some Object Oriented Languages, a class can have more than one super class. A class is analogous to a template from which objects are "derived".

An object class **inherits** characteristics from its super classes. Inheritance is a powerful concept that enables a subclass to inherit all the methods of its super class, by default, unless the subclass chooses to provide new methods that override the defaults. This allows new classes to be created from old classes.

Real-time object-oriented model

The requirements for a real-time object-oriented model differ from those of a non real-time object-oriented model in two important ways:

1. The object model should support the encapsulation, abstraction, and understanding of the object's functional and temporal characteristics.

2. The implementation of the object model (classes, instances, methods, and messages) and the associated run-time mechanisms should have predictable temporal characteristics.

The real-time object model extends the object-oriented model to describe real-time properties in programs. In this model, active objects with timing constraints describe a system, together with their interaction through message passing. Such an object is called a *real-time object.* The real-time object will execute its method on receipt of a message from another real-time object within the environment. The method execution time needs to be clearly defined through profiling runs or by estimation techniques. Similarly, as with method execution the message passing mechanism's underlying implementation must guarantee the timeliness requirement. Therefc when designing objects, we must be able to express real-time attributes such as deadlines, start-times and severity of objects.

Since the object is an extension of an abstract data type, it should encapsulate its use of time. This allows the implementation of an object A to be changed without modifying the objects with which A interacts. However, this requires that A's externally-visible behavior be invariant, including both its functional behavior, and its temporal behavior (e.g. the execution time of A's operations). Therefore, we have the problem of preserving invariant temporal behavior of an object as seen by other objects, while permitting variations in object's internal timing. Although proper techniques of object-oriented design ensure that changes in the application environment, will not affect an object's functional interface, they offer no such assurance for an object's external temporal interface. Consequently, when an object's implementation is changed, often the temporal behavior of all objects that interact with it may have to modified.[7]

In a real-time system different activities execute at different priorities, depending on the severity and the timing requirements. The object-based system must reflect these priorities, and furthermore, as objects invoke other objects, the original priority information should be used in scheduling the invoked object to maintain a consistent priority scheme across the entire system. The object invocations must carry some sort of priority information in order to achieve this priority propagation.

The object granularity of traditional object-oriented systems ranges from fine-grained objects the size of individual data items like integers and reals to large-grained objects that would correspond to heavyweight processes. Since real-time systems require very fast response and high efficiency, the system designer must strike a balance between using high-overhead, fine grained object model in comparison with large-grained objects which require relatively low-overhead. In addition these objects must encapsulate both data and threads of control.

Predictable, dynamic creation and destruction of objects is necessary for real-time systems. A traditional object-oriented model supports the creation of objects as needed by the application and the destruction of objects when no longer needed The Real-time object model should ensure that this flexibility does not alter the system's timeliness requirement.

RELATED WORK

One model that was developed for real-time systems based on the object paradigm is the ROOM[12] (Real-time Object Oriented Model) methodology. This methodology provides a framework to systematically capture, design and implement the functionality of a real-time application. It defines a cohesive and consistent set of powerful abstractions intended primarily for event-driven, real-time, distributed systems[14]. It permits the designer to define a real-time system in terms of multiple, concurrent objects. The following is a list of key characteristics of this methodology :

1. ROOM provides a development framework to systematically capture, design, analyze and implement a real-time application. The ROOM development process is based on the iterative view of software development, where the process is broken down in terms of activities, differentiated on the basis of their respective objectives. It takes place in three phases: the analysis phase, the design and implementation phase, and the execution and verification phase. During each of these phases, a model is built and progressively enriched. The model is then verified by executing on a simulator , and the behavior is compared against a set of prepared test cases.

2. ROOM consists of a modeling language, a compiler and a virtual machine which can execute the compiled modeling language. The modeling language contains a formal syntax and is different from a traditional programming language, in that, it uses different formalisms (syntax) at various abstraction levels and modeling dimensions. The authors claim that because the programming concepts are formally related, it is not possible to generate an inconsistent system model[14] using this language The virtual machine, on the other hand, permits the compiled model to be executed and enables the designer to graphically view it on the screen. There are two implementations of the ROOM virtual machine available : 1. The ObjecTime simulator 2. The ObjecTime MicroRTS.

3. ROOM supports the existence of concurrent objects. In ROOM, an *actor* is a concurrent object that exists (and can execute independently of) other actors in the same environment.

These objects can communicate with each other as long as a binding exists between them. Actors in ROOM are instances of actor classes, and the actor classes can be organized into inheritance hierarchies, thereby allowing entire system architecture to be refined and reused through standard inheritance mechanism. This provides a major boost to productivity and reliability.

4. ROOM supports the run-time instantiation of actors. Actors are instantiated by their containing actor as the system runs. Once it has been instantiated, a dynamic actor runs concurrently with its containing actor. This provides flexibility to the real-time system designer to replace existing actors with special purpose actors, during emergency situations.

5. ROOM expresses the behavior of an object (actor), to an external event using a variant of the statechart formalism known as ROOMcharts[13]. This description consists of listing the states, the transitions, and defining the events that trigger an internal transition. A state transition causes an event to be processed. The actual details of the event handling can be written in two different object-oriented languages, namely C++ and a special derivative of Smalltalk. The states, transitions, and all of the event handling schemes follow the same pattern in that the subclass inherits from the parent class. ROOMcharts have been made available to the world through a commercial toolset called ObjecTime which supports the ROOM methodology.

The key advantage of the ROOM methodology is its capability to execute the design model at an early stage thereby enabling the designer to gain early insight into the requirements/design problems. Secondly, the availability of concurrent, dynamic objects enables the designer to create several task oriented objects with specific time deadlines. Therefore, in the case of an emergency situation, where certain objects fail to meet their deadlines, they can be replaced by other objects during run-time. Thirdly, once the implementation is complete, and the system is deployed, all changes are made by modifying the ROOM design model, instead of modifying the source code. This preserves the original architecture, thereby avoiding architecture decay. However, one of the disadvantages of this methodology is lack of formalism during the specification of the design/requirements model. The availability of formal syntax is not enough to analyze models for consistency and completeness.

A CASE STUDY

We consider an example of a railroad crossing system. This example has been extracted from Achuthan et al[1]. In this paper, an object-oriented view of the problem with an emphasis on its real-time reactive behavior is provided. The problem is nontrivial and is adapted to focus on the modularity achieved through object-orientation; the problem also forms an adequate basis for discussing instantiation, aggregation, inheritance and sub typing.

An informal Description

A railroad crossing system consists of a collection of *trains* and a collection of *gates* servicing the roads crossing the train tracks. The gate should remain closed when a train goes past the crossing. In order to control the gates there is a collection of *controllers* such that one controller is associated with each gate. A controller closes its associated gate when it gets a "nearing signal" from a train and opens the gate once all the trains crossing the gate have left. A controller does this by receiving signals from the trains and transmitting necessary control signals to its associated gate. The problem is general in the sense that more than one train can cross a gate simultaneously, probably, through multiple parallel tracks; a train can independently choose the gate it is going to cross, probably based on its direction/zone of travel. A safety requirement for the system is that whenever a train is inside a gate, the gate should remain closed.

When a train wishes to cross a gate, the train begins by sending a *Near* message to the controller associated with the gate. Similarly, while leaving the gate, the train informs the controller by an *Exit* message. A typical time constraint on the train is that there is a delay of at least 2 units before the train gets into the gate after it sends the signal *Near*. Furthermore, after the message *Near*, there is a maximum delay of 5 units before which the train should exit the crossing.

A controller upon receiving a *Near* message from a train, signals its associated gate to *Lower*. Similarly, following an *Exit* message from the last train to leave the gate, the controller sends a *Raise* message to the gate. There are two time constraints associated with the controller. The controller should respond:

1) with a *Lower* message to the gate within 1 unit of time after receiving the *Near* message.
2) with a *Raise* message to the gate within 1 unit of time after receiving the *Exit*

message from the train to leave the gate.

A gate responds to a *Lower* message and a *Raise* message by closing and opening the gate, respectively. There is a minimum delay constraint of 1 unit for closing the gate and a minimum and maximum delay constraint of 1 and 2 units, respectively, for opening the gate.

In the next subsection, the notation used to describe the real-time requirements of the different objects in the system is presented. This is followed by a formal specification of the system using the **TROM**^{2} model.

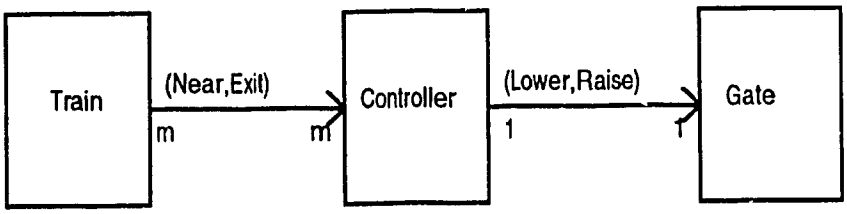


Figure 2 : Class Interaction diagram

Specification using the TROM model

This section describes the notation used in the **TROM (Timed-Reactive Object Model)** model to specify the system requirements. Objects in **TROM**^{2} are represented in terms of class definitions. The header consists of the keyword **Class.** Following the header is the **Event** section, enumerating the set of events associated with the class. The input events and output events are marked by the symbols '?' and '!', respectively. The internal events are left unmarked. The **State** section, enumerates the abstract states of the class together with the state hierarchy. The states prefixed with '*' are the initial states. The substates (if any) of a state are enumerated within parentheses following the state. The **Attribute** section describes the set of attributes belonging to the class together with their types. This section is followed by the **Trait** section. The **Trait** section imports the behavior of the data model *Set* defined elsewhere using Larch Shared Language(LSL). Thus the **Trait** section defines the link between the two tiers i.e., the data abstraction and the class definition.

The **Attribute Function** defines the association of attributes to the states, thus partitioning the domain of attributes into active and dormant parts at each state. Syntactically, the statement *S1,S2,S3 --> x*, means that the attribute *x* is the only active attribute in the states S1,S2, and S3. The **Transition** section describes the transition specification for each transition in the state machine. The **Time-constraints** section describes the time constraints associated with various transitions.

A Formal Model

The interacting objects in the case study basically consists of three types of components *Train, Gate* and *Controller.* The instantiation relationship in object-orientation helps to specify the system using three class specifications one for each of the above component types. The class interaction diagram for the system is shown in Figure 2. An object configuration diagram for the system is shown in Figure 3.

The class specifications of each of the train and controller classes together with their state diagrams are shown in Figure 4. A train starts in state S1. The attribute *cs* of S1 denotes the set of controller objects known to the train. The port condition of the *Near* event in a train specifies that the train can non-deterministically select a controller from the set *cs* for interaction By doing so, a train object models its intention to cross the gate associated with the controller. The attribute *cr* of states S2, S3 and S4 denotes that further interaction of the train should be with the controller it chose, until it exits the gate. The internal events *In* and *Out* signifies the start and end, respectively, of the action of crossing a gate by the train. The post-condition of the transition associated with the event *Exit* specifies that the set of controllers known to the train remains unchanged. The two time constraints associated with a train are specified by the two tuples. The trigger event for both the time constraints are *Near.*

Initially, the controller is in state C1. The attribute *kt* denotes the set of train objects with which the controller can interact and *gt* denotes the gate associated with the controller. A controller can receive events from multiple trains. When in state C1, the controller responds to the input event *Near* from a train by sending the event *Lower* to the gate within 1 unit of time. This corresponds to the approaching signal from the first train to enter the crossing after the gate was last opened. Subsequent *Near* events mark the approaching signal from other trains, probably in parallel tracks. The attribute *inSet* associated with the states C2 and C3 denotes the set of trains in the crossing. It is obvious from the post conditions that, an insertion into the *inSet* and a deletion from it, is done by the transitions *Near* and *Exit*, respectively. Once the controller receives the

event *Exit* from all the trains which were crossing, the controller will send the event *Raise* to the gate within q time units. The port specification for the events *Near* and *Exit*, indicates that for each train instance any *Exit* event should be preceded by a *Near* event and every consecutive *Near* event should be interleaved by an *Exit* event.

The class specification for the gate class is shown in Figure 5. The gate is open in start state G1 and closed in state G3. Interaction of the gate with its controller is through the events *Lower* and *Raise*. The attribute *cr* in the gate denotes the controller with which the gate is associated. The events *Down* and *Up* are internal events and denote the end of the actions, closing and opening, of the gate, respectively. The reactions associated with the two time constraints of the gate are triggered by the events *Lower* and *Raise*, respectively.

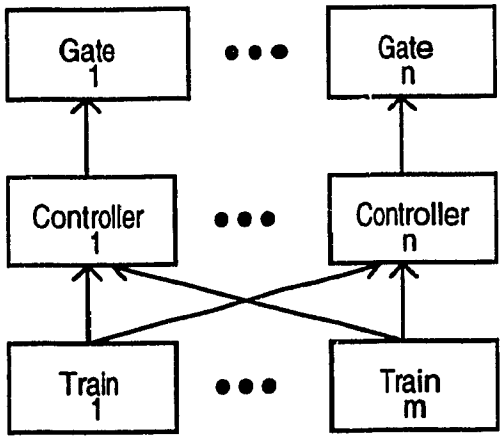
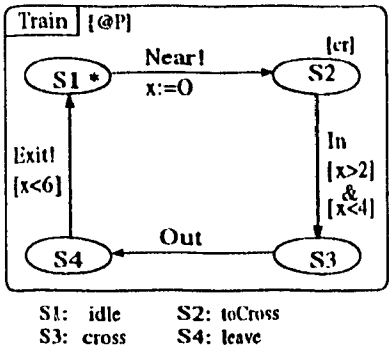


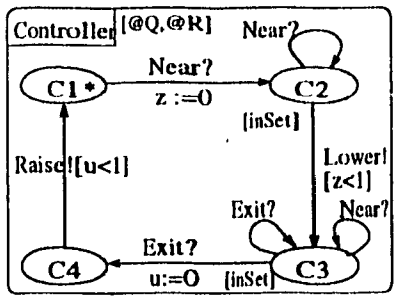
Figure 3 : The object configuration diagram

Class *Train* [@P]
  Events: $Near!, Exit!, In, Out$
  State: $*S1, S2, S3, S4$
  Attributes: $cr : @P$
  Attribute-function:
    $S1, S3, S4 \mapsto \{\}$; $S2 \mapsto cr$;
  Transition Spec:
    $R_1 : (S1, S2)$; $Near!(\mathsf{pid} : @P)$; $true \Longrightarrow cr' = \mathsf{pid}$;
    $R_2 : (S2, S3)$; In; $true \Longrightarrow true$;
    $R_3 : (S3, S4)$; Out; $true \Longrightarrow true$;
    $R_4 : (S4, S1)$; $Exit!(\mathsf{pid} = cr)$; $true \Longrightarrow true$,
  Time-constraints:
    $(R_1, In, [2,4], \{\})$
    $(R_1, Exit, [0,6], \{\})$
end



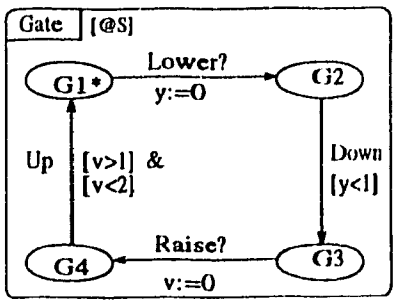
S1: idle S2: toCross
S3: cross S4: leave

Class *Controller* [@Q,@R]
  Events: $Near?, Exit?, Lower!, Raise!$
  State: $*C1, C2, C3, C4$
  Attributes: $inSet : TSet$;
  Traits: $Set[@Q, TSet]$ /* Link to LSL tier */
  Attribute-function:
    $C1 \mapsto \{\}$; $C2, C3, C4 \mapsto \{inSet\}$;
  Transition Spec:
    $R_1 : (C1, C2)$; $Near?(\mathsf{pid} : @Q)$;
      $true \Longrightarrow inSet' = insert(\mathsf{pid}, inSet)$;
    $R_2 : (C2, C2), (C3, C3)$;
      $Near?(\neg(\mathsf{pid} \in inSet) \wedge (\mathsf{pid} : @Q)$;
      $true \Longrightarrow inSet' = insert(\mathsf{pid}, inSet)$;
    $R_3 : (C2, C3)$; $Lower!(\mathsf{pid} : @R)$; $true \Longrightarrow true$;
    $R_4 : (C3, C3)$; $Exit?(\mathsf{pid} \in inSet)$;
      $(size(inSet) > 1) \Longrightarrow inSet' = delete(\mathsf{pid}, inSet)$
    $R_5 : (C3, C4)$; $Exit?(\mathsf{pid} \in inset)$;
      $(size(inSet) = 1) \Longrightarrow inSet' = delete(\mathsf{pid}, inSet)$;
    $R_6 : (C4, C1)$; $Raise!(\mathsf{pid} : @R)$; $true \Longrightarrow true$;
  Time-constraints:
    $(R_1, Lower, [0,1], \{\})$
    $(R_5, Raise, [0,1], \{\})$
end



C1: idle C2: activate
C3: monitor C4: deactivate

Class *Gate* [@S]
  Events: $Lower?, Raise?, Down, Up$
  State: $*G1, G2, G3, G4$
  Transition Spec:
    $R_1 : (G1, G2)$; $Lower?(\mathsf{pid} : @S)$; $true \Longrightarrow true$;
    $R_2 : (G2, G3)$; $Down$; $true \Longrightarrow true$;
    $R_3 : (G3, G4)$; $Raise?(\mathsf{pid} : @S)$; $true \Longrightarrow true$;
    $R_4 : (G3, G4)$; Up; $true \Longrightarrow true$;
  Time-constraints:
    $(R_1, Down, [0,1], \{\})$
    $(R_2, Up, [1,2], \{\})$
end



G1: opened G2: toClose
G3: closed G4: toOpen

Figure 4: Class specifications in TROM for Train, Controller and Gate

Design of the Railroad Crossing system

The Railroad Crossing system is a good application where all the fundamental characteristics of real-time systems are represented. The design of this system comprises modeling the real-time functions of a controller that monitors a particular railroad crossing. The main responsibility of this controller is to ensure that all trains pass through only when the gate in that intersection is closed. The controller must coordinate the arrival of the trains in the crossing and control the position of the gate. Given this outline, we shall enumerate the characteristic features of this system. The controller being the central element receives input from a collection of trains and provides output to the gate. In a real situation, the input to the controller, will be obtained from the environment when a train passes sensors located several kilometers from the entry and exit of the crossing. The controller must be capable of processing inputs from multiple trains concurrently, that is, it should be designed to monitor as many number of tracks as the intersection designer desires. The controller maintains information about the status of its tracks. It must respond to messages received from trains with guaranteed response times and ensure the safety of the crossing. These response times are intentionally kept very small in this application to verify the effectiveness of the design. In the event of a system failure, the software must raise an exception and provide warning messages on the screen. The reliability of the system is very important since failure to meet the real time deadlines can result in severe casualties to the population using the Railroad crossing.

In this section, we describe the design of this system. We begin by discussing the role of the formal requirement specification, and its contribution to the design of the system. Section II presents the criteria based on which the system was designed. This is followed by an introduction to the design of key objects, their attributes, and the services provided by each one of them. Section IV describes the time parameter, how each object guarantees the prescribed response times and what action is taken in failing to meet the time constraints. Since the system was implemented in Borland Pascal, which is a non-concurrent programming language, an event scheduler was designed, to simulate concurrency. Section V presents the components of the scheduler, and outlines the mechanism by which events are dispatched to the appropriate objects. A description of the system user interface, is provided in Section VI describing how instances of trains and controllers are created during run-time, and the features provided to verify and test the system. The report concludes with an analysis of the design.

Design Based on System Requirements

The Railroad crossing system was designed based on the formal specification provided using **TROM(Timed-Reactive Object Model)**. This formal specification, described the temporal characteristics, the reliability requirements and the desired behavior of the software to real-time events. It consisted of two parts. The first part described the properties, the responsibilities, and the actions of each agent in the application, using a formal notation It encapsulated the object states, the events and the system timing requirement for each event. In addition, it formally listed the functions of an object by stating the pre and post condition for each action The second part modeled the state changes due to events and graphically represented the deadlines to respond to each event. Finite state diagrams were used to represent the state changes for each object in the application. This specification model was a starting point for the preliminary implementation design of the system.

Almost all computing projects start with an informal description of what is desired It is at this stage that the functionality of the system is defined. There are three notations used to express the system requirements : 1) informal , 2) structured, and 3) formal. Informal methods usually make use of natural language and various forms of imprecise diagrams They have the advantage that the notation is understood by a large group of people (that is, all those speaking the natural language). However, the greatest disadvantage is that this representation is open to number of different interpretations. Structured methods often use a graphical representation but unlike the informal diagrams these graphs are well defined. Although the structured methods are quite rigorous in their approach, they cannot, in themselves, be analyzed or manipulated For such operations to be carried out, the notation used to capture the behavior needs to have a mathematical basis. Methods that have such mathematical properties are known as formal These techniques have the clear advantage that precise descriptions can be made in these notations. Moreover, it is possible to prove that the properties specified in the model hold. The disadvantage with them is that they cannot be easily understood by those not prepared or able to become familiar with the notation. The high reliability requirement in real-time systems has caused a movement away from informal approaches to the structured and increasingly, the formal.

The **TROM** model is a formal technique designed to express the time-dependent requirements and design of real-time systems. It is particularly useful to model event-driven systems The Class construct captured the properties of the different objects in the Railroad crossing application, thereby introducing the aspect of modularity in the specification. This mapping of the agents to real-world objects provided a practical representation of the object responsibilities, and

outlined the services provided by each one of them. Since the specification provided a object-oriented view of the problem, the transition from design specification to implementation was very smooth.

The behavior of objects in the Railroad system is based on the stimulus applied to them. This was modeled in the **TROM** specification as finite state machines in which a response at any instant is determined by the object's present state and the stimulus applied at that state. The event-driven nature of the Railroad application required modeling the set of interactions between the different objects. In addition, since these interactions were governed by timing constraints, the model captured the inter-object interactions through their state diagrams and associated a timing constraint for each event (internal or external) that caused a state change. Finite state machine based specification was extremely useful to capture the dynamic behavior of the system. This event based analysis made the specification very intuitive by providing a view of the system as responding to a set of events, rather than a sequential list of application requirements. Therefore, the **TROM** formal notation coupled with the **TROM** graphical presentation (Finite State machines) provided an accurate understanding of the application domain, free of implementation details, which is the foremost characteristic of a quality specification.

Design Criteria

During the design of this real-time software, there emerged a requirement to establish a set of standards or criteria. This was required to aid in the final evaluation/analysis of the overall system. The most important stage in the development of any real-time system is the generation of a consistent design that satisfies a specific set of requirements. Since the applicability of the OO paradigm to the design of real-time system was the topic of the project, it was important to establish the type of deliverables expected from the design stage. There are several of these criteria ranging from design simplicity to design efficiency. However, four of them were considered very important to the design of real-time systems. They were : *correctness, modularity, verifiability, and ease of use.*

Correctness is the ability to exactly perform the tasks, as defined by the requirements and the specification. The code produced for the system, when inspected, should ensure that it correctly implements the design and has not introduced any errors.

Modularity is the ability to produce modules that could be reused, in whole or in part, for new applications. These objects must communicate with as few others as possible, they should exchange as little information as possible, and information in the module must be private to the module unless it is specifically declared public. The internal methods in the module must hold together. A good software design encompasses highly cohesive modules that are loosely coupled[11].

Verifiability is the ability to generate test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases. Verification accomplished by testing should demonstrate that the result of executing each logic branch, each input/output statement, and each logic segment in general satisfies the specification requirements.

Ease of Use is determined by the ease of learning how to use the software, preparing input data, interpreting results and recovering from usage errors. The design of the screens should be unambiguous and data entry should be accommodated with the minimum number of keystrokes.

The following sections present the design of the various components of the Railroad crossing monitoring software.

Objects, Attributes and Methods

In the Railroad crossing system the problem is decomposed into areas of responsibility, each realized by an object. The idea was to simulate a real world situation by placing independent objects in the crossing to do the work needed to get the job done. Like real-world objects, the independence of the objects enables the design of the application to be conceptualized in terms of cooperating entities, not just as one solution program. Thus the application is essentially a collection of these cooperating objects.

The agents in the crossing were designed as descendants of the Turbo vision TView object (Figure 1). For example, the controller object is a descendant of TWindow object. It inherits all features of a Window (such as window frame, close icon etc.) and adds controller specific display strings. The Train, and Gate object are designed using the same approach. On the arrival of an external event, each object sends internal messages to update its display with the data supplied in the events. This places the responsibility on each object to refresh its display. The Track object, on the other hand is defined as a descendant of TInputline which is also a descendant of

TView. This object is created on the initialization of the controller object and is owned by the controller. In addition to responding to events to update its display, the Track object contains capabilities to respond to mouse events (Single Click, Double Click etc.) thereby enabling future extensions to the object.

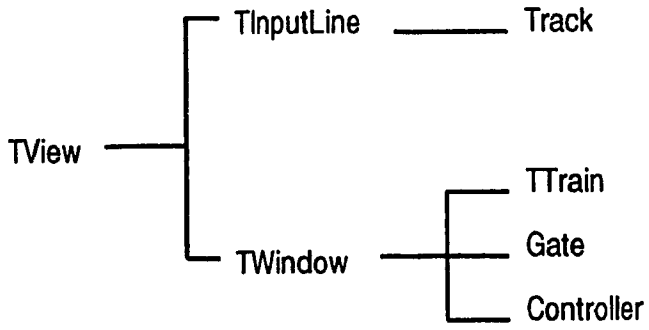


Figure 5 : Railroad application class hierarchy

The most important data in this application is the real time constraints of each object. Each object maintains its proper copy of the data. Each object communicates with the other object by sending and receiving messages. A message may request some service or supply status or any kind of information. All messages generated by an object are typically stored in an event queue. The requests are scheduled by a central event scheduler by processing the event queue. A description of the Train, Gate, Controller, and Track objects is provided below.

Train Object

Responsibilities:

On the receipt of a "GO" message from the central event dispatcher the train must enter, cross, and exit the gate at the specified time interval. The train must also reflect any change in state by updating its display.

Attributes:

Trainid	Unique train number
TrainState	Four possible states : *Near_Gate, Crossing, Exiting, Traveling*
Entry_MinDelay	Minimum delay before entry into crossing
Entry_MaxDelay	Maximum delay in the crossing

Methods

Constructor *TTrain.Init*

Inherits the TWindow Init { Initializes the window frame, background, etc.. }

Initializes all attributes, and instantiates the train

Procedure *TTrain.handleEvent*

Inherits the TWindow HandleEvent { this handles all window related operations }

Handles Command **GO, NEAR_GATE, CROSS_GATE, EXIT_GATE**

Controller Object

Responsibilities

Process messages received from all the trains wishing to cross the gate in a timely manner. Assure delivery of message to the Gate object within the specified deadline. Ensure safety of trains in the crossing by verifying that the tracks are free prior to raising the Gate. Refresh controller display by updating the track status on entry and exit of trains.

Attributes

ControllerState	Current Operating State (*Normal, Error*)
Controllerid	Unique controller number
Gate	Pointer to the Gate object in the crossing

Tracklist	List of tracks monitored by controller
Lower_MsgDelay	Maximum delay before lower msg sent to Gate
Raise_MsgDelay	Maximum delay before raise msg sent to Gate

Methods

Constructor *TController.Init*

Inherits the TWindow Init { Initializes the window frame, background, etc.. }

Initializes all attributes, the list of tracks, the gate, and instantiates the controller

Procedure *TController.HandleEvent*

Inherits the TWindow handleevent { this handles all window related operations }

handles commands from the train { Train_Near, Train_Exit }

sends commands to the gate { Lower_Gate, Raise_Gate }

Function *TController.TracksClear : Boolean*

Returns true if all the tracks are free

Destructor *TController.Done*

Inherits the *TWindow Done*; { removes the memory allocated to window objects }

Remove the memory allocated to the Gate object

Remove the memory allocated to the Track object

Gate Object

Responsibilities

Raise/Lower the Gate on receipt of the message from the controller within the specified time interval. Provide a display of its status i.e. Gate Up, Gate Down.

Attributes

GateState	Two possible states : Up or Down
Gateid	Unique Gate number
Down_MsgDelay	Maximum delay for lowering the gate
UpMsg_MinDelay	Minimum delay prior to raising the gate
UpMsg_MaxDelay	Maximum delay prior to raising the gate

Methods

Constructor *TGate.Init*

Inherits the *TWindow Init* { Initializes the window frame, background, etc.. }

Initializes all attributes, and instantiates the Gate. This routine is called by Controller Init.

Procedure *TGate.HandleEvent*

Inherits the TWindow HandleEvent { this handles all window related operations }

Handles commands from the controller { Lower_Gate, Raise_Gate}

Track Object

Responsibilities

Maintain information about the current state of the track (Busy, Free) and who currently owns it.

Update its display based on the change of events.

Attributes

TrackNumber	Unique track number for each controller
TrackState	Two possible states : *Busy or Free.*
Who	Current owner of the track (ex. Train # 3)

Methods

Constructor *TTrack.Init*

Inherits the TInputline handleevent

Initializes all attributes and instantiates the track

Procedure *TTrack.Evaldata*

Displays the Current status of the track on the screen (Free, or Train #)

Procedure *TTrack.HandleEvent*

Inherits the TInputLine HandleEvent

Handles two commands : TrackBusy, TrackFree

These are received from the controller when the trains enter and leave the crossing.

On receipt of the messages, the track updates its internal state and it external display.

Temporal Interactions between the objects

The Railroad crossing application required two types of behaviors from its objects with respect to timely execution of an action The first type of behavior expected is the performance of actions within a given interval of time relative to the occurrence of an event. In this case, the timing constraints are specified relative to the current time, called *now,* in units of hours, minutes, and seconds. The second type of object behavior is to perform actions at fixed points in time, and the interval at which some action has to take place is specified with accuracy relative to a chosen granularity. This time is represented as a wall clock time (ex. Eastern Standard time) (year:month:day:hour:min:sec:msec). The scheduling of new trains at fixed intervals of time by the Application object, is an example of this kind of timing behavior.

The Railroad crossing application is paced with reference to a external global clock which provides a sequence of discrete ticks This global clock is used as a real-time base, and it is of a defined granularity (the difference in time between any two consecutive ticks). It is turned on when the application object is initialized (*init_sys_clk*). All the objects synchronize their tasks in reference to this global clock. A routine (*Get_sys_time*) returns the time elapsed since the clock was started. One of the first applications of this routine is in *TApplication.Idle:* to schedule trains on tracks at fixed intervals of time. When the Idle routine executes, it checks whether it is time to schedule new trains. If it is time, new "GO" event objects are created and inserted into the event queue for execution. The event objects contain the time at which they must be executed. The time specified in the object is the absolute time of the global clock. The Event scheduler (*GetEventFromQ*) also uses the global clock to dispatch events from its queue. Since the Railroad system is entirely event-driven, the information used to schedule the different events in the system is the scheduled time of each event object.

The other type of behavior guaranteed by the objects is the execution of an action within the prescribed time bounds relative to the occurrence of an event. An event is any action that can change the object's state; e.g., execution of a statement, sending of a message, receiving of a message. Events can originate from an object itself, another object, or the external world. Each object also guarantees, to raise an exception when a task's deadline is missed. The object provides this guarantee by stamping the time on receipt of a message from another object. The time stamp denotes the time of arrival of the event, at which point a local timer is started. If a timer expires prior to the complete handling of the event, the object raises a flag, updates its internal status, and updates its external display.

Encapsulating the time parameter within an object is one of the features of the real-time object oriented model. In real-time systems true object-oriented design should capture time in the same manner as data and functionality. This allows the implementation of an object A to be changed without modifying the objects with which A interacts. The Railroad crossing objects encapsulate their timing constraints for each event. The strength of the OO model is to contain the temporal constraints into understandable, manageable groups. The kinds of time constraints encapsulated within the objects are as follows:

- Maximum - No more than *t* time units may elapse between two events.
- Minimum - No less than *t* time units may elapse between two events
- Duration - An event or a sequence of events must occur for *t* time units.

Modifying the Event Scheduling Mechanism

The real world is full of concurrency. A typical Railroad crossing consists of multiple tasks containing several trains crossing the gate simultaneously. The controller is expected to handle multiple requests to service all trains wishing to cross The initial implementation of the system highlighted two major problems:

1. the controller was designed as a single-threaded object handling requests in a First Come First Serve basis
2. the order in which the events were dispatched to all the objects was also on a First Come First Serve order. This created a priority inversion problem, making even the highest priority request wait for the completion of all existing requests.

The initial train scheduling routine consisted of a central loop dispatching "Go" messages to all the trains in random. On receiving the "Go" message, the train would send a "TRAIN-NEAR" message to the controller advising it of its intention to cross the gate within 2 seconds. The Controller would in turn send a "Lower" message to the Gate and do the necessary operations of updating its current status. This mechanism worked fine for one train, one controller scenario. However, when more train objects were instantiated and the responsibility of the controller was augmented to concurrently handle more than one track, the controller would process the requests from trains in a sequential manner from entry to exit, that is, it would process a TRAIN-NEAR message, lower the Gate, wait for a TRAIN-EXIT message, Raise the Gate, and then process the next TRAIN-NEAR message and so on. In other words, as the controller was processing a TRAIN-NEAR message, it would not release control to the central scheduling system to handle

other messages that might have been sent to it. The second problem was caused by the Turbo Vision event gathering mechanism. This routine services keyboard, mouse, and other events at the same priority level as the application object events. This First Come First Gather policy caused the objects in the Railroad crossing application to miss their deadlines, since the scheduling mechanism could not guarantee delivery of messages to the destination at the prescribed time. This required the Turbo Vision event scheduling system to be modified to accommodate for the real-time nature of the Railroad application.

Prior to describing how the event scheduler was modified it is appropriate to examine the event gathering, and event scheduling mechanism provided as part of the Borland Turbo Vision package. At the very highest level, the main program of the Railroad crossing application consists of three statements:

```
var
   ARailCross : TApplication;              { declare an instance of the TApplication object }
begin
   ARailCross.Init;  { initialize the ARailCross object }
   ARailCross.Run;{ interact with the user }
   ARailCross.Done;          { dispose of the ARailCross object }
done;
```

The first of the three statements (ARailCross.Init) is the constructor call to create the *TApplication* object and it sets up the main program for use. As a convention, all Turbo Vision constructors are named *Init*. *Run* is where the application gathers events, and routes events to the appropriate objects. It consists primarily of a **repeat..until** loop, shown here in pseudo-code format:

```
repeat
    Get an event;
    Handle the event ;
until Quit;
```

In essence, the application loops through two tasks: Getting an event, and servicing that event. The *GetEvent* task looks around and checks to see if anything has happened that should be an event. If it has, *GetEvent* creates the appropriate event record. The *HandleEvent* task then routes the event to the proper objects. Eventually, one of the events resolves to some sort of Quit command, and the loop terminates. The *Done* destructor disposes of the objects owned by the application. It reverses the actions of the *Init* constructor.

Turbo Vision Events

Events in Turbo Vision can be best described as little packets of information describing discrete occurrences generated in an application. Each keystroke, each mouse action, and any of certain conditions generated by other component of the program, constitute a separate event. Events are not objects, but a record structure that convey information between objects.

At the core of every event record is a single field named *What*. The numeric value of the *What* field describes the kind of event that occurred, and the remainder of the event record holds specific information about that event; the keyboard scan code for a keystroke event, information about the position of the mouse and the state of its buttons for a mouse event, and so on. All these events are stored in a FIFO (First in First Out) structure.

Turbo Vision GetEvent

The different objects in the application need not worry about where the events actually come from. The *GetEvent* method is the only routine that concerns itself with source of the events. Objects in the application simply call *GetEvent* and rely on it to take care of reading the mouse, the keyboard, and the pending events generated by other objects. As shown below in pseudo-code format, the *GetEvent* loop scans among the mouse, and the keyboard and then calls *Idle*. *Idle* is another *TApplication* method that is executed if no event is ready. It is a method that is called by *GetEvent* when no events are being generated by the application. The real time clock is updated in the *Idle* method.

```
begin
   Get any Pending Events;
  If No Pending Events then
    Get a Mouse Event
    If No Mouse Event then
        Get a Keyboard event
        If No keyboard event then
            Idle;
        end if;
    end if;
end if;
end;
```

The Second source of Event records

The previous section probably gave the impression that event records are always obtained in a the *GetEvent* method in *TApplication*; but there is a second way an event record can get through to a *HandleEvent* method. A special function exists in Turbo Vision for a view to directly call the *HandleEvent* of another object. This function, known as *Message*, calls the relevant *HandleEvent* directly, without going via *GetEvent*. This is extremely useful when the receiver of the message is known and the programmer wishes to bypass the processing overhead of the *GetEvent* method. In our case, the controller on receiving the TRAIN-NEAR message, must immediately send a "Lower" message to its gate. This is done using the message function. The format of the message function is shown below.

message (Receiver,ACOMMAND,evCommand,DATA)

Real Time Event Scheduler

As presented earlier, the main program processing loop gets an event, and routes the event to the appropriate object. The deficiency with this loop for our application is that there is no concept of time associated with the event. We are unaware of the time at which the event was generated, and the bounding time within which the event must reach the destined object. In addition, all events are handled with the same priority. For example, if a "TRAIN_EXIT" message is sent to the controller object by a train, and simultaneously if the user decides to move a train "window" from one location to another creating several mouse event messages, the delivery and execution of the "TRAIN_EXIT" message cannot be guaranteed with the existing dispatching mechanism.

Two modification were introduced to the existing mechanism. Since the Turbo Vision event is not an object, it couldn't be overridden, therefore a new event object (EventRec) was created. This object is an extension of the event record with the time variable. Also, the event queue was modified from the current FIFO arrangement to a priority queue system where events are stored in the order in which they must be scheduled. Finally, the TApplication GetEvent method was overridden to provide higher priority to the command events over usual Turbo vision events. The following sections describe each one of the above changes in detail.

Event Object

The event object is created by an object when it wishes to communicate with another object at a specified time. The event object is created by calling the *Init* method, at which point the sender initializes the pointer to the destination object, the command to be executed, the time at which the event must be scheduled, and any data (if any) to be passed to the destination object. The event object is then inserted into the Event collection to be scheduled by the modified *GetEvent* method. The receiver on handling the event, calls the done method to release the memory allocated to the event. The new event object is shown below. Since the train, controller and gate are modeled as window objects, the source and destination parameters is of Pview type, which is a parent class of the TWindow object. The data field is a record structure which allows objects to pass information between each other.

```
PEventRec = ^TEventRec;
TEventRec = object(TObject)
        Source : PView;
        Command : Word;
        Destination : PView;
        Data : TMsgFormat;
        Time : Longint;
        constructor init( Asrc:PView;Acmd:Word;Adest:PView;
                          Dat : TMsgFormat;Tim:longint);
        destructor done; virtual;
       end;
```

Event Queue

The Turbo Vision Event Queue was a FIFO structure with a maximum limit of 16 events being stored at any point in time. A new event queue structure (*TEventCollection*) was created. This structure stores all the event objects in an ascending order sorted by the time of execution, that is, the event that needs to be scheduled closest to the current time is at the head of the queue. The new event queue is implemented as a descendant of a Turbo vision object, *TSortedCollection*. *TSortedCollection* is an abstract object which automatically sorts the new members added to the collection. In order to use this feature, the descendant object must override two methods: *Keyof* and *Compare*. The *Keyof* method specifies which field of the objects in the collection must be used as a sort key and the *Compare* method determines the sort order by comparing two clients and deciding which one belongs ahead of the other in the collection.

TEventCollection knows how to insert events and delete existing ones. When an event has to be added to the queue, the collection's *Insert* method is called (*Queue^.Insert(Event)*). The *Insert* method relies on *Compare* to determine where the event should be placed in the queue. The *Insert* method implements a binary search through the collection's items using *Compare* to compare the items. The sort key is the time field of the Event object. In order to allow duplicate objects, i.e. two events to be scheduled at the same time, the *Duplicates* attribute of *TSortedCollection* is set to true. So if an event needs to be inserted into the collection and another event already exists with the same time value, the Insert method inserts the new event before the first existing event. The code for the Compare function is shown below.

```
function TEventCollection.Compare;
begin
    if (PEventRec(Key1)^.time < PEventRec(key2)^.time) then
            compare := -1   { Insert Key1 object before Key2 object }
    else    if (PEventRec(key1)^.time > PEventRec(key2)^.time) then
            compare := 1    { Insert Key1 object after Key2 object }
    else
            compare := 0;   { Insert Key1 object before Key2 object }
end;
```

Modified GetEvent

The *TApplication.Getevent* method being the routine that concerns itself with the source of the events, had to be modified to handle the priority inversion problem. This method fetches the events from the event queue. This method needs to be overridden in order for it to fetch higher priority (scheduled) events first, and if none exists then fetch regular mouse and keyboard events. The overridden *GetEvent* is shown below. It first calls function *GetEventfromQ* which verifies if an event needs to be scheduled at the current time and returns it if one exists., This event is immediately dispatched to the destination object (*Desktop^.handleEvent*). If no event exists, the function returns an event object with a constant *evNothing* in the *Event.What* field, in which case the inherited *TApplication GetEvent* is executed.

```
procedure TRailCross.GetEvent (var Event : TEvent)
begin
    GetEventFromQ(Queue,Get_Sys_time, Event);
    If Event.What <> evNothing then
    begin
            desktop^.HandleEvent(Event);
    end
    else
            inherited GetEvent(Event);
end;
```

User Interface

The Railroad crossing system user interface was designed to satisfy four basic requirements:

1. provide run-time instantiation of trains, controllers, and tracks
2. display all messages exchanged between objects
3. enable run-time modification of real-time constraints to help during testing
4. provide visual and auditory display in case of failures

The library routines provided with the Turbo Vision object-oriented framework helped immensely in quickly creating the graphical user interface. The main display consists of three visible components : the menubar, the desktop, and the statusline (Figure 6). The menubar highlights the program name and provides a display of the real time clock. The desktop, on the other hand, is the area where the results of all the actions requested takes place. The statusline appears at the bottom of the screen and it reminds the user of basic keystrokes and shortcuts applicable at that moment in that active window. It allows the user to click the shortcut keys to carry out the action.

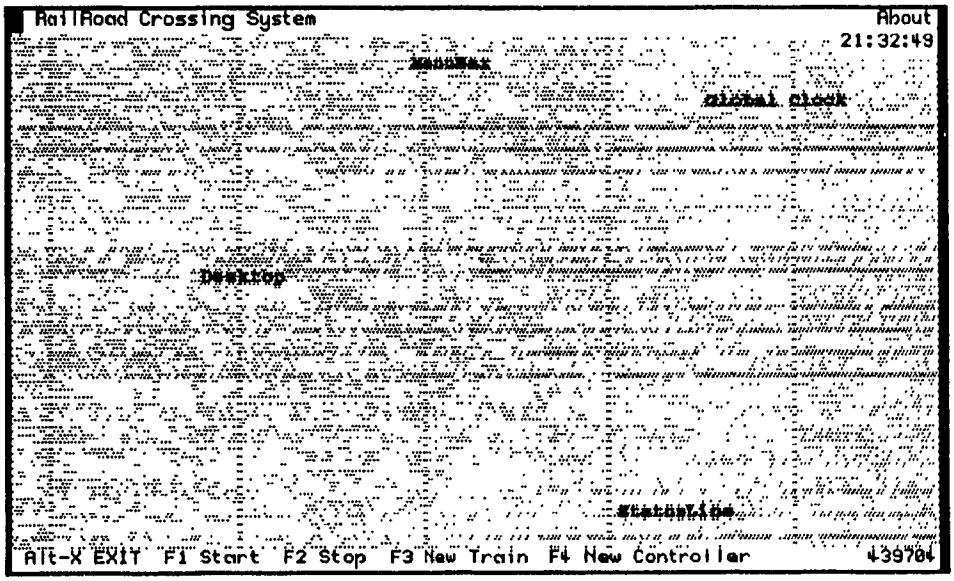


Figure 6 : Railroad Crossing System Main Display

Run-time Instantiation of Objects

The system allows creation of new trains and controllers by either clicking on the status line with the mouse or by pressing F3 or F4 respectively. The Controller and Gate objects are always created as a pair. The train, controller, and gate objects are descendants of the Turbo Vision TWindow object. Thus each instantiation creates a window. When a controller is instantiated, the number of tracks being monitored by the controller can be specified. The appropriate number of track objects will be instantiated and will be inserted into the controller window. The train, controller, and gate objects respond to all of window commands (move, zoom, etc..). On top of the window the number of the object is specified. Figure 7 displays three train instances, and one controller/gate monitoring three tracks.

The track object, on the other hand is descendant of TInputLine, and is owned by the Controller. It is a strip of information being displayed in the controller window. It responds to update messages from its owner. The user can also double click on this strip and the object can be enhanced to perform other functions. An example of a future enhancement might be to display the status of a track (Active, Defective...) on demand. By double clicking on the appropriate track the strip can send a message to a physical track monitor device, obtain additional information, and display it to the user.

Each object displays the messages it sends and receives to other objects. They also display their current state in the status field. The Train has three possible states : Traveling, Crossing, or Exiting. If the train is Crossing a controller, the track # allocated to it at the Railroad crossing is displayed. The controller is usually in Normal state. In case of a error, i.e. failure to meet the real time constraints, it switches to an error state. The Controller window also indicates the status of its tracks. The Gate has two states: Up or Down. It displays all messages received from its controller.

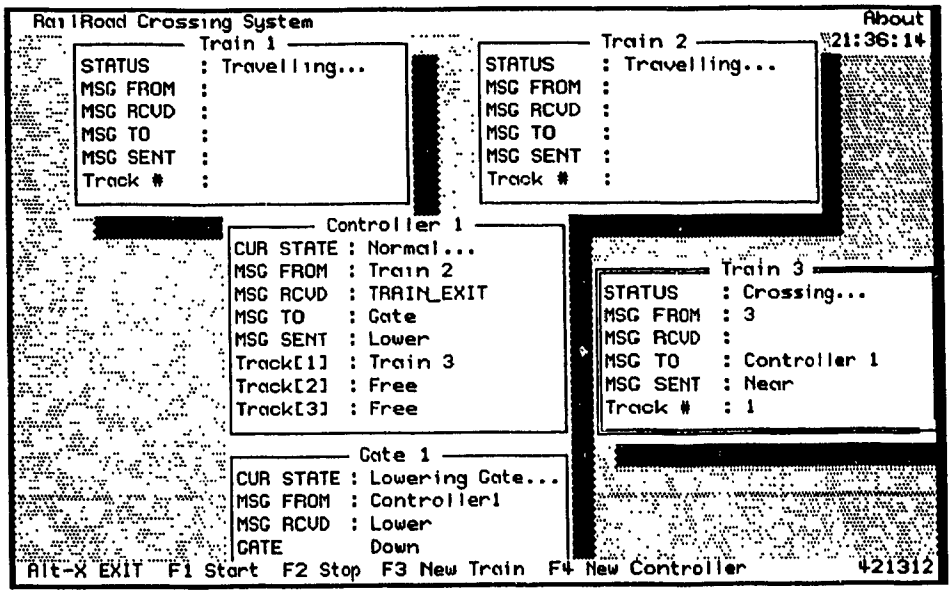


Figure 7 : Instance of Train, Controller, Gate, and Tracks

Modifying Real-time Constraints

During the verification phase of the software, it was necessary to supply the system with proper inputs to check whether it satisfies the properties or conditions specified in the formal specification. This involved introducing intentional delays or merely changing the real-time constraints to test whether the desired result was obtained. In addition, this dynamic entry of inputs should not interfere with the simulation process, i.e. the trains must still cross and the controller must perform its functions while the inputs are being entered. Figure 8 shows the case where different parameters of the Gate 1 object are being displayed. This is obtained by double clicking on the Gate object. Once modified, the new values will take effect.

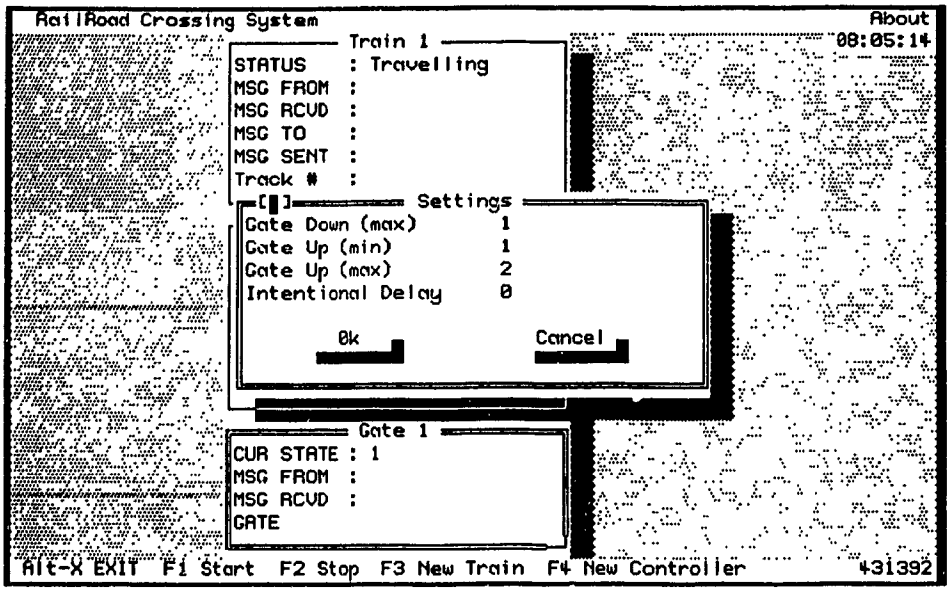


Figure 8 : Modifying the Gate Object Real time settings

The implementation of a hard real-time system has to guarantee that all deadlines will always be met. In the case of a failure, the ideal system has to provide feedback to the user on its status. The objects in the Railroad crossing system provide a visual and auditory display in the case of a system failure. Figure 9 displays the case where, the controller is unable to meet the required timing constraints. This results in both the trains crossing the gate when the Gate is Up.

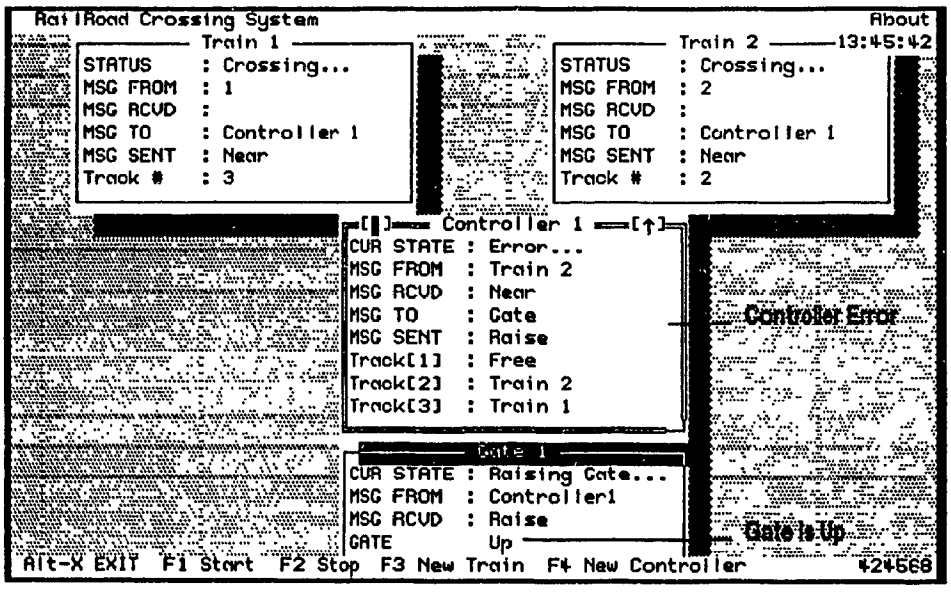


Figure 9 : Controller Failure

Design Analysis and Conclusion

The design of the Railroad crossing system was implemented using the object-oriented component of Borland Pascal 7.0, also known as Turbo Vision. Turbo Vision is the library which was used to write Borland/Turbo Pascal 7.0 itself. The Railroad application currently consists of 450 lines of code. Most of the objects to display data were inherited from the Turbo Vision libraries, and some of their methods were overridden This demonstrated the power of the OO paradigm to reuse and customize object behavior for designing a new application. This paradigm was appropriate for the current application, which resulted in the creation of a natural design The Railroad system consists of a collection of trains which interact with controllers which in turn interacts with gates in a common environment. The OO paradigm has a similar view in which the world is viewed as a collection of discrete objects which act and react in a common environment. This is quite different from the conventional way of designing in which the system is decomposed based on the functions to be performed by the software system. The result of object-wise decomposition is that when the characteristics of entities such as Train, for example, is to be modified, it is done in only one place, i.e. where the Train Class is defined. In the functional approach, one has to examine all the functional modules, as the behavior of a train is spread over all the modules that use trains. The most important differences between object-oriented and conventional analysis methodologies ultimately stem from the object-oriented requirements of encapsulated operations. A functional decomposition of systems violates encapsulation because operations can directly access a multitude of different entities.

There is a smooth transition from design to implementation in the Object Oriented paradigm. The property of encapsulation makes it possible to have software objects that directly correspond to the physical entities such as Train, Controller, etc.. Therefore, the entities that the user and the software engineer discuss, when defining the requirements as part of the project, are the same entities the designers build objects from and programmers work with during implementation of the requirements. Therefore, object oriented design makes implementation easier by improving the task of communication between users, designers and implementors. Another side-benefit of this approach is that it encourages more time to be spent in thinking about the conceptual design of the system. The main emphasis is identifying the similarities among objects if the system designer has to take advantage of the inheritance mechanism. The use of objects and message therefore allows an easy, understandable system design to be developed. The **TROM** design specification provided a precise understanding of the application design, hence enabling a smooth transition to implementation.

Once classes are defined in a object-oriented domain, they are used repeatedly in building new software in the domain. For example, once we define the classes Train, Controller, Gate, and Track, if we need to produce more intelligent Gates or Tracks, it can be done by extending the original classes. The new controller could inform the new trains in the case of a system failure, thereby stopping the train from entering the crossing. The result is a different approach to software development : it is application development by the assembly and refinement of existing proven objects, rather than by writing code from scratch. Another important payoff of good object-oriented programming is that it facilitates change naturally. Once the technology is successfully adopted, we find it easier to keep pace with changing user needs.

One of the major strengths of the OO paradigm is the inheritance concept. This feature encourages reuse of existing code by allowing definition of application-specific object descriptions (*subclasses*), based on previously defined generic object descriptions (*classes*). The implementation of the Railroad crossing system in a real world situation will consist of interfacing the objects to the external world through a particular hardware medium (for example, RS-232). Assuming that all the objects are controlled using the same medium, the implementation of the communication protocol can be encapsulated in an abstract superclass, known as *Devices*. The Device class will then be common specification for all class instances that use the same communication protocol. The subclasses can inturn inherit from this generic specification. The possibilities to enhance the current design using the OO paradigm is immense. However, adoption of this technology requires a long term view. It is a strategic investment that yields over the long haul.

My opinion is that the OO paradigm permits the designer to quickly model and implement the basic functionality of the target system. It then allows for gradual improvements to the software such as adding error recovery, making use of sensory information, and speeding up performance. In the long run when a library of useful objects has been created, the upgrade and maintenance of applications is very simple. As it is well accepted in Computer Science, that getting autonomous objects, programs, people or whatever to work together is a difficult problem, a mechanism for cooperation among objects (hardware and software) and people towards a common goal is an important research topic in this area.

REFERENCES

1. R. Achuthan, V.S. Alagar, T. Radhakrishnan, "A formal model for the Object-Oriented development of Real-time Reactive Systems", Workshop on Object-oriented Real-time systems - OOPSLA94, Portland, Oregon, October 1994.

2. R. Achuthan, V.S. Alagar, T. Radhakrishnan, "Object-Oriented Specification of Reactive Systems", Tech. Report, Concordia University, May 1994.

3. Gomaa.H, "Software Design Methods for Concurrent and Real-Time Systems", Massachusetts, Addison-Wesley Publishing Company, Inc, 1993.

4. Jacobson. I, "Object-Oriented Software Engineering", Riverside Printing Co. (Reading) Ltd, 1992.

5. Mercer.W & Tokuda.H, "The Arts - Real-time Object Model", In proceedings of the IEEE 11th Real-time systems symposium, Lake Buena Vista, Florida, pp. 2-10, (1990).

6. Nirkhe.M.V, Tripathi.K.S, Agrawala.K.A, "Language Support for the Maruti Real-time System", In proceedings of the IEEE 11th Real-time systems symposium, Lake Buena Vista, Florida, pp.257-265, (1990).

7. Stankovic.A.J and Ramamritham.K, "Tutorial - Hard Real-time Systems", Washington, Computer Society Press of the IEEE, 1988.

8. Ward.T.Paul and Meller.J.Stephon, "Structured Development for Real-time system", Vol. I, New York: Yourdon Press, 1985.

9. Rine C. David and Bhargava. Bharat, "Object-Oriented Computing", Computer- October 1992, pp-6-10.

10. Bihari.T, Gopinath.P, Honeywell, "Object-oriented Real-time Systems: Concepts and Examples", Computer - December 1992, pp.25-32.

11. K. Nielsen, K. Shumate, "Designing Large Real-time Systems with Ada", New York, Intertext Publications/Multiscience Press, Inc, McGraw-Hill Book Company, 1988.

12. S. Gheorghe, J. McGee, "Using Object-Oriented modeling to design complex real-time embedded systems", Real-time Engineering Computing With a Deadline, Volume 1, Number 4, Winter 94.

13. B. Selic, "An efficient object-oriented variation of the statecharts formalism for Distributed Real-time systems", in Proc 5th International workshop on CASE, Montreal, Canada, 1992.

14. B. Selic, G. Gullekson, J. McGee, and I. Engelberg, "ROOM: An Object-Oriented Methodology for Developing Real-time systems," in Proc 5th International workshop on CASE, Montreal, Canada, 1992.

15. Bihari.T, Gopinath.P, Schwan.K, "Object-Oriented Design of Real-time Software", In proceedings of the IEEE Real-time systems symposium, Las Alamitos, California, pp.194-201,(1989).