A METHODOLOGY FOR OBJECT-ORIENTED MODELING AND DESIGN OF REAL-TIME, FAULT-TOLERANT SYSTEMS

DEBERA R. HANCOCK

A METHODOLOGY FOR OBJECT-ORIENTED MODELING AND DESIGN OF REAL-TIME, FAULT-TOLERANT SYSTEMS

by

Debera R. Hancock

A Thesis Submitted to the Faculty of the

College of Engineering

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Engineering

Florida Atlantic University

Boca Raton, Florida

May 1997

A METHODOLOGY FOR

OBJECT-ORIENTED MODELING AND DESIGN

OF REAL-TIME, FAULT-TOLERANT SYSTEMS

by

Debera R. Hancock

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. E. B. Fernandez, Department of Computer Science and Engineering, and has been approved by the members of her supervisory committee. It was submitted to the faculty of the College of Engineering and was accepted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering.

SUPERVISORY COMMITTEE:

[illegible]

Thesis Advisor

[illegible]

[illegible]

Jim Wu

Chairperson, Department of Computer Science and Engineering

[illegible]

Dean, College of Engineering

[illegible]

Dean of Graduate Studies and Research

4/4/97

Date

ABSTRACT

Author:	Debera R. Hancock
Title:	A Methodology for Object-Oriented Modeling and Design of Real-Time, Fault-Tolerant Systems
Institution:	Florida Atlantic University
Thesis Advisor:	Dr. E. B. Fernandez
Degree:	Master of Science in Computer Engineering
Year:	1997

Many methodologies for software modeling and design include some form of static and dynamic modeling to describe the structural and behavioral views respectively. Modeling and design of complex real-time software systems requires notations for describing concurrency, asynchronous event handling, communication between independent machines, timing properties, and accessing real time. Function-oriented structured analysis methodologies such as Ward and Mellor's SA/RT and Harel's Statecharts have provided extensions for real-time system modeling. Dynamic modeling of real time systems using object-oriented methodologies also requires extensions to the traditional state machine notations in order to convey the real time system characteristics and constraints. Shaw's Communicating Real Time State Machines (CRSM's), Harel's O-Chart notations, and the Octopus methodology provide methods for modeling real-time systems consistent with object-oriented methods. This thesis proposes an object-oriented analysis and design methodology that augments the traditional Object Modeling Technique (OMT) dynamic model with real-time extensions based on high-level parallel machines and communication notations from CRSM. An example of the proposed methodology is provided using a realistic but hypothetical example of an automated passenger train system. A design refinement step is included for fault tolerant considerations. An evaluation of the proposed methodology with its extended notations is provided.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

1. Introduction

Real-time systems are characterized by their real-time response requirements (deadlines) and underlying concurrency of functions. These time-critical systems often have stringent safety requirements, necessitating that they be highly reliable and that their functions be predictable when subjected to real-time, concurrent events. Although the systems may be quite complex, good analysis and design methodologies must be simple and understandable while conveying accurately the design and its real-time aspects.

Methodologies for specifying system requirements and designs typically include some form of requirements specification and notations for modeling the static, dynamic, and functional aspects of the system. The requirements specification is often a textual description of the functional characteristics of the system. Although text descriptions are desirable since they are easy to read by non-programmers, text descriptions alone may not serve to accurately and unambiguously define the requirements. Formal languages are required to clearly specify and validate critical system requirements such as timing and safety constraints [Heit 95]. Formal languages also provide a mechanism for validation of the design and implementation with respect to the requirements. For real-time systems where there are often critical timing requirements or safety considerations, verification of the design and implementation with respect to formally specified critical requirements is necessary.

Once the requirements are specified, a series of analysis and design steps are performed that refine and map the requirements to a complete design. With object-oriented methods, the analysis step includes a representation of the real-world problem as a static class diagram. The initial static representation of the system closely maps to the real-world elements and should not include elements introduced for design or implementation [Rumb 94A]. This modeling of the real-world problem into independent, data-encapsulated classes maps conceptually into the system as a collection of concurrently active communicating components. In a good methodology, the mapping of the problem information described in the requirements specification to the objects in the static class diagram should be consistent and visibly intuitive. Although the object-oriented analysis model of the system conveys the objects of the system and their high-level relationships, the class diagram does not model any of the system behavior, concurrency, or timing of events that are critical to the specification of real time systems.

In practice, the complete system design is typically derived through a series of refinements from the static analysis model. It is often difficult to pinpoint when analysis ends and design starts. A series of iterations of lower level analysis and design steps is performed. Many notations and methodologies have been used for these steps. With any good methodology, some form of dynamic modeling is required to model and design the behavior of the system elements over time. State machines and variations on state machine notations are popular modeling tools for both object-oriented and functional methodologies. High level state machines are often decomposed and refined into lower-level state machines. Event trace diagrams and scenario descriptions are examples of design approaches that are used to refine the operations and states of the lower level state machines. For object-oriented design, the initial static class diagram is further refined to include implementation constructs, packaging features, and optimizations. Throughout each iteration, the mapping from one refinement to the next should be easily traceable for design verification. Similarly, the real-time properties and constraints of the system should also be traceable through each refinement of the design.

1.1 Motivation

Many real-time method extensions for function-oriented methodologies have been proposed and applied to real system development [Ward 86][Goma 86][Hare 96B][Hare 90][Farn 96][Leve 94]. Object-oriented design has recently become popular as an alternative for designing complex software systems. It is clear that the popularity and inherent concurrency of object-oriented designs makes it a highly desirable approach for producing real-time systems. Revised methodologies that map closely to object-oriented methodologies, programming languages, and implementation tools have been developed such as O-Charts [Hare 96A], Real-Time Object-Oriented Modeling (ROOM) [Kras 95][Seli 92], and the Octopus methodology [Awad 96]. Each methodology has its own unique notations and approach. Although the industry has not standardized on a particular object-oriented modeling methodology, Rumbaugh's Object Modeling Technique (OMT) [Rumb 91] is generally well known and understood by object-oriented designers and developers. OMT is a comprehensive methodology, encompassing the entire analysis and design life-cycle with modeling of the static, dynamic, and functional views of the system. The popularity of OMT has proven it to be a sound approach for developing software systems. Extending OMT for real-time systems preserves the well-understood and readable approach of OMT while introducing extensions necessary for clearly modeling real-time systems. The methodology extensions proposed in this thesis are based upon state machine notations that are also well known in the industry, thus preserving the readability of the resulting models and designs.

The dynamic modeling in the standard OMT [Rumb 95] does not provide a mechanism for visibly distinguishing the physical system concurrency and external events from the concurrency introduced by the underlying object design. OMT also does not clearly distinguish external communication mechanisms from internal design-oriented communication. This thesis proposes modifications to the standard OMT dynamic modeling to include a hierarchical system design based on concurrent high level (physical) communicating state machines. The notations for OMT state diagrams are modified and extended to include notations found in Communicating Real-Time State Machines (CRSMs) [Shaw92] and OMT notation extensions proposed by M. Chonoles and C. Gilliam [Chon 95]. The resulting modeling and design approach follows the general approach for OMT, with the addition of the CRSM machine-level diagram and real-time notation extensions for communication and timing.

1.2 About This Thesis

Section 2 provides an overview of the proposed new object-oriented modeling and design methodology. Section 3 provides background information on Object Modeling Technique (OMT) which is the basis of the proposed methodology extensions and modifications. An overview of other proposed object-oriented and function-oriented methodology extensions for real-time systems is also provided in section 3. Sections 4 through 8 use the proposed methodology to model and design a hypothetical automated Passenger Train System (PTS). These sections contain example specification elements, analysis model, dynamic model, design refinements, fault-tolerant design modifications, and implementation considerations. A summary and evaluation of the proposed methodology, along with a process diagram, is provided in section 9. Section 10 provides the thesis conclusions and considerations for future study.

2. Proposed Methodology

A seamless development approach is utilized from analysis through design as described in [Rumb 94]. In a seamless approach, boundaries between steps are not rigid. From the specification of the requirements, the problem progresses from model to detailed design through a series of refinements that use the same or consistent notations, syntax, and constructs. Details, optimizations, and implementation considerations are added iteratively. From each step to the next, there is a clear mapping of the higher level requirements or design elements to the lower level refinements. This is essential in order to verify that the resulting design and implementation is correct. This section describes the process steps proposed by this thesis for modeling object-oriented real-time systems. Section 4, "Passenger Train System (PTS) Requirements Specification", on page 20 provides an example model and design using this methodology with a realistic but hypothetical system.

2.1 *Requirements Specification*

The approach used for specifying the system functional requirements is a combined formal and informal specification. The majority of the functions are specified using problem statement text and illustrations to provide a functional description of the requirements. For critical safety and timing constraints and system state information, a more formal property-based language is used to describe the requirements. Although more algebraic and less readable than the text descriptions, the formal specifications provide more precise and verifiable information about the real-time system behavior with respect to critical system events or states. Use cases are also included as a part of the requirements specification. These use cases define the black box (externally visible) behavior of the system and are used in later process steps to determine key event-trace scenarios.

2.2 *Object Oriented Analysis (OOA)*

The analysis phase used is identical to the traditional OMT analysis phase [Rumb 91][Rumb 94A] . An object (class) diagram for an analysis model is intended to describe the real-world problem, not necessarily the design or implementation of the solution. Only attributes, operations, and associations are illustrated in the analysis model where they are necessary for understanding the classes and their relevant relationships. The OOA class diagram is an abstraction of the system in that it helps in understanding the model by suppressing detail and providing an overall view of how the system elements fit together. The system may be abstracted at an even higher level into subsystems with relationships defined between each subsystem [Rumb 94B].

2.3 *Dynamic Modeling with Communicating Real-Time State Machines*

This thesis proposes that the traditional dynamic modeling step from OMT [Rumb 91][Rumb 95] be augmented with a multi-level communicating state machine hierarchy based on Shaw's two-level Communicating Real-Time State Machines (CRSMs) [Shaw 92]. Additional notation extensions [Chon 95] [Rumb 95] are utilized where necessary to clarify the real-time nature of the system. The effect is a highly readable combined set of notations utilizing the strengths of each approach. Since the notations are conceptually consistent they can be easily combined. The following notation conventions are proposed in conjunction with the dynamic modeling steps in this methodology:

- Use "[condition]" notation for event conditions from OMT instead of the "condition ->" notation from CRSM. This is necessary to avoid confusion with event counting described below.

- Use "<command>?" and "<command>!" notations for external I/O from CRSM instead of the "^target.sendEvent(arguments)" notation from OMT. The explicit input and output notations crisply specify the nature of the communication and the paired interaction between concurrent state machines.

- Use the real-time notations "RT(x) [y]" from CRSM to specify timeouts and the reading of real time. This is a necessary feature for specifying real-time systems.

- Use internal command notations from CRSM. Internal commands can take the form of operations with arguments or operations such as assignment ("x=y+z"). Multiple internal commands are separated by semi-colons. This notion is simple and flexible.

- Use shorthand naming conventions and event nomenclature from [Chon 95]. The shorthand names simplify the specification and diagrams. The names used in the Passenger Train System example are described in the example specification in section 4.3.

- Ambiguous events are referred to with qualifiers of the form "Event.qualifier" [Chon 95]. This is necessary in order to represent identical instances of machines or objects (as in the trains in the Passenger Train System example).

- The notion of an event counter (reset by "counter->0") and the specification of the I-th event (Event Label, I) is also used [Chon 95]. When used as a constraint, the I-th event notation is enclosed In square brackets, "[(Event Label,I)]". This notation simplifies the state diagrams.

- States are illustrated as circles instead of the traditional round-cornered rectangles. This is done for ease in producing the diagrams with standard drawing tools.

The proposed dynamic modeling step contains hierarchical communicating state machines. The level 1 model depicts the high level communicating machines. Levels 2-n provide the modeling and design of the finite state machines that execute on each machine.

2.3.1 Level 1: High Level Communicating Machines

The first step is to provide a CRSM high level machine diagram illustrating the physically concurrent machines and channel communication between the machines. This high-level state machine approach maps easily from the real-world elements (or subsystems) of the OOA class diagram. It also clearly illustrates the physical separation of independent machines and controllers and their inherent parallelism. In object-oriented analysis, all objects are conceptually independent and concurrent. Traditional object-oriented design notations quickly mix the objects that are introduced for design with the objects representing controllers in the real-world. The addition of the first level CRSM in the hierarchical approach visibly distinguishes the physical parallelism from the object design concurrency. Similarly, the external communication (over CRSM channels) between machines is easily distinguished from design-introduced communication between objects.

2.3.2 Levels 2-n: Low Level State Machines

As in CRSM, the next step is to expand each high level machine into lower level state machines. The proposed approach modifies the OMT state machine notations to provide CRSM low level finite state machines with internal and external commands. These low-level state machines are very similar to the original OMT dynamic model with relatively few notation changes. These state machines should be easily understood by persons familiar with traditional state machine notations. Like OMT, the states in each CRSM can be composite states that are further refined by additional state machine diagrams with expanded levels of detail. This is preferable to Shaw's CRSM two-level hierarchy which could result in highly complicated state machines at level-two for complex systems.

2.4 Iterative Design Refinement with Event-Trace Diagrams

In practice, it is difficult to find a clear distinction between when modeling stops and design begins. Initial models and designs are often modified or corrected when the details of specific scenarios are examined.

The system designer typically derives the design through an iterative process of switching design focus from high level to low level and back again. From the initial high level model, specific use case scenarios are used to validate the high level model and develop the details of the lower level. Since each scenario is only a particular sequence of events that could occur, they are not comprehensive. However, they do serve as a good tool for refining the detailed model and design.

Event trace diagrams based on the use cases from the requirements specification provide a mechanism for illustrating a particular scenario and the associated events between objects for that scenario. The proposed methodology uses event-trace diagrams as a tool for refining the high level machines into refinements of lower level state machines and transitions (events or communication). From the very first high level machine, the lower level machines and communication (internal and external) are clarified and refined using event-trace diagrams.

2.5 *Fault-Tolerant Design with Failure Scenarios*

The base design is developed first using the combined OMT-CRSM methodology with extensions for real-time notations as described above. The focus of this initial base design is on the normal operation of the system to maintain the operational and safety requirements. As such, elements of fault avoidance and fault detection are built into the base system design and are reflected in the behavior of the key controllers of the system. However, the initial base design derived in this manner is not necessarily fault tolerant.

Single hardware and software component failure scenarios are then applied to the base system design to determine the critical components for failure detection and safe operation. Any necessary redundancy to further detect and mask critical faults (fault tolerance) is added to the design at this stage. An abstraction hierarchy of the system illustrates the task concurrency, software versions, and process mapping to multiple hardware systems.

3. Background

This section provides a brief overview of the Object Modeling Technique (OMT) that is the underlying methodology upon which the proposed extensions and modifications are based. A key extension suggested by this thesis is the introduction of Communicating Real Time State Machines (CRSMs) into the OMT dynamic modeling step. An overview of CRSMs is also provided.

Many object-oriented and function-oriented methodologies and extensions have been proposed in software engineering literature. Examples of these are included in sections 3.3 and 3.4. This thesis does not provide a comprehensive overview of existing methodologies, only representative examples.

3.1 Object Modeling Technique (OMT)

This thesis proposes extensions to Object Modeling Technique (OMT) as the base methodology for developing the object oriented analysis and design. The basic OMT methodology and notations are described in detail in [Rumb 91].

Using the OMT methodology, the first step in analyzing the problem is constructing an object model, referred to as the analysis model. The object model depicts the state (data) of the real-world problem and organizes the problem into workable, understandable pieces. The analysis model should capture the information describing the problem without implementation details. Key relationships between objects are shown where they serve to clarify the problem or requirements. Key attributes are shown where they are important to understanding the state of the objects or relationships between objects.

Dynamic modeling is utilized to further clarify the problem and define the temporal behavior of the objects in the model. An overview of dynamic modeling is provided in [Rumb 95]. Event trace diagrams and scenarios illustrate the ordering of a particular set of interactions (message flows) that occur over some time period between specific instances of objects. Any real time constraints can be illustrated effectively in the event trace diagram. Text scenarios, naming conventions, and notations are helpful in further understanding the problem. The event trace diagrams are useful in identifying states and events for the state diagrams.

State diagrams are generated for key objects (controllers). The state diagram describes how the object behaves when any of all possible events are received from other objects or external agents. The state diagram formalizes the pattern of events, states, and state transitions. State diagrams for various classes can be combined, illustrating their interaction, via shared events. State diagrams are particularly useful in assisting with the identification of operations (methods) for objects.

The OMT methodology includes a functional modeling step that is not discussed or used in this thesis. The functional model describes the computations within a system. Data flow diagrams are used to show the flow of values from external inputs, through operations and internal data stores, to external outputs.

In addition to any refinements made to the object model as a result of the previous modeling steps, the next step in OMT design is to further refine the design to take into account more implementation-level considerations. Modularity, concurrency, and packaging are considered along with data encapsulation and module interdependencies. The system may be divided into modules or subsystems with a subsystem diagram indicating the relationships and dependencies between them.

OMT can be used for modeling and design of controllers and real-time systems [Rumb 95][Rumb 93]. Composite objects containing an aggregation of several parts are inherently concurrent processes with individual state machines. In addition to explicit modeling of parallelism by aggregation, object instances themselves can be thought of as concurrent processes, each with a distinct dynamic behavior in response to methods (operations). Object states can be also be modeled as superstates with concurrent substates (implementation threads). The sending of external events is modeled as objects sending events to other objects as part of the specification of transitions. Internal events are modeled as operations. Timing considerations can be illustrated in event trace diagrams and as constraints on transitions.

Real-time notation enhancements to the OMT dynamic model for shorthand naming conventions, event counters, and timing constraints were provided by Chonoles and Gilliam in [Chon 95]. Rational Inc. has defined real-time extensions to Unified Modeling Language (UML) [Rati 96], but their extensions are limited to time markers in scenarios (event trace diagrams). Another extension of the same type is given in [Cole 92]. These are mostly notational extensions, appropriate only for analysis. This thesis proposes a more comprehensive set of process steps and extensions that include support for the other life cycle stages.

3.2 Communicating Real-Time State Machines (CRSMs)

Communicating real-time state machines (CRSMs) are a notation for specifying concurrent, real-time systems including monitoring and controlling functions [Shaw 92]. CRSMs complement the OMT notation of state machines in the dynamic modeling step. CRSMs have the notion of synchronous communication between state machines instead of the OMT notation for external events. CRSMs also have added notations and facilities for describing timing properties and for accessing real time.

CRSMs are useful in specifying real-time systems because of the inherent parallel behavior of real-time system controllers. The notations for clearly illustrating this parallelism, the communication between parallel objects, and the timing elements make CRSM a powerful extension to OMT for real-time systems.

CRSMs are specified in two levels. The first level is the machine level. The machine level concept is similar to the OMT notion of concurrent subsystems and interaction between subsystems [Rumb 94B]. CRSM machines at the machine level execute concurrently and independently of one another, except when they directly communicate. Machines are distributed (they do not share variables other than the notion of real time). Machines communicate over channels that connect pairs of machines. A channel is an abstraction for a directed connection between two machines. Channels are also uniquely identified with an event or message. A separate channel is associated with each different communication type between machines. The channel notation provides a mechanism for arguments or message components that are the content of the communication.

Each machine from the first level is further refined in the second level as a serialized finite state machine. These state machines are conceptually similar to OMT state machines although the notations for event transitions differ. The CRSM state machines have guarded internal commands instead of constrained event transitions. The CRSM guards are equivalent to the OMT constraints. CRSM commands can be internal or external commands. External commands are I/O commands where "<command>?" is an input command and "<command>!" is an output command. These external commands are conceptually similar to external events in OMT notation although the notations for input and output in CRSM more crisply identify the type of communication than in OMT (where the event name implies the type of communication). CRSM internal commands are equivalent to transition events and operations in OMT.

CRSM provides notations for specifying real-time functions and properties. An external real time clock command of "RT(x)? [y]" results in a state transition and generates a timeout at relative time y, setting x to the real time at the time of the timeout. When the timeout notation is omitted, "RT(x)? [0] or RT(x)?", the result is that x is set to the current real time. When the time argument is omitted, "RT? [y]", a pure timeout is generated at relative time y. The CRSM real-time notations are useful for specifying periodic events and timeouts for real-time constraints.

3.3 Real-Time Extensions to Function-Oriented Methodologies

This section provides a sampling of real-time extensions to function-oriented methodologies. There are many function-oriented methodologies, some of which have been practiced for over ten years. This section highlights only a few.

3.3.1 Transformation Schema

Notation extensions to data flow diagrams were proposed by Ward [Ward 86] to specify the timing and control interactions in systems. This extended data flow diagram, referred to as the transformation schema, provides visual model distinctions between data and control in the transformations and data flows. Transformations and flows associated with controllers and events are designated with dotted lines. Event types and directions are distinguished by the type and direction of arrow heads. Transformation and token-based execution rules define the state of the system over time. The movement of tokens provides the modeling of time units, delays, and synchronization.

3.3.2 Activity Charts and Statecharts

Activity charts are hierarchical data-flow diagrams specifying the functions (activities) of the system and the data that flows between functions. Statecharts are a graphical dynamic modeling language for specifying the behavior of the functions over time. Statecharts were originally introduced by D. Harel in 1987 [Hare 87] and are based upon a function-oriented, structured analysis methodology. The language of activity charts and statecharts is commercially implemented in the STATEMATE system [Hare 90]. The definition of statecharts has evolved over time with review and use. Updated semantics and issues are described in [Hare 96B].

Statecharts are extended state machine diagrams. There are three types of states in a statechart: OR-states, AND-states, and basic states. Unlike traditional state machines, concurrent state machines can be graphically illustrated with the AND composition. Concurrent state machines are separated in the same diagram with dashed borders. When a state containing AND-states is entered, each of the parallel state machines is entered. All state machines are exited when any transition is taken out of the parallel state. Similar to OMT dynamic modeling, statechart diagrams have also have transitions, events, guard conditions, and actions. In statecharts, concurrent (AND) state machines can be synchronized by these transitions. Statechart notations also include methods for specifying the real-time scheduling of actions, timeout events, and event broadcast.

The behavior of statecharts is described by a set of possible runs, containing a series of steps representing the system status with respect to a sequence of external events. Two models of timing are supported in the STATEMATE implementation: synchronous and asynchronous. The synchronous time model assumes that the system executes a single step in every unit of time, reacting to all external events that occurred since the previous step. The asynchronous time model is more realistic for modeling real-time systems since multiple steps can execute at the same point in time in response to external events.

3.3.3 Modecharts

Modecharts [Farn 96] are a specification language for real-time systems that are based upon a hierarchical composition of serial and parallel modes (similar to states). Serial modes are similar to traditional sequential state machines or statechart OR-states. Parallel modes (statechart AND-states) are completely concurrent and independent. Transitions between modes in parallel are not allowed. The semantics of modecharts are defined in terms of Real Time Logic (RTL). The modechart language provides mechanisms for precisely specifying deadlines and delays. System properties are expressed in RTL notations. The hierarchical organization of modes and the corresponding specification of RTL properties at higher levels of abstraction is intended to provide a mechanism for property verification of large systems.

3.3.4 Requirements State Machine Language (RSML)

Requirements State Machine Language (RSML) [Leve 94] is a specification language based on statecharts and modified for practical use in developing complex real-time applications. RSML extensions to Harel's traditional statecharts include directed communication notations (instead of broadcast communication), distinction between internal and external events, AND/OR tables for state transition specification. RSML uses a time model similar to STATEMATE's asynchronous time model where an RSML state machine takes all transitions triggered by an external event followed by the transitions triggered by the subsequently generated events.

3.4 Real-Time Extensions to Object-Oriented Methodologies

This section provides a sampling of real-time extensions to object-oriented methodologies. With the growing popularity and practice of object-oriented methodologies and tools, several real-time extensions have been proposed. Many seem oriented towards implementation tools and therefore closely reflect the underlying implementation language (C++ or Ada). The following section provides background information on a subset of the existing real-time extensions to object-oriented methodologies.

3.4.1 O-charts

O-charts are defined by Harel [Hare 96A] as the object-oriented equivalent to Activity Charts and Statecharts. O-charts are an OMT-like language for describing the structure of object-oriented classes and their relationships. Statecharts (with object-oriented extensions) are used in conjunction with O-charts to specify the behavior of O-chart objects. O-charts are consistent with the popular OMT concepts. With the

O-chart model, objects generate events which are placed in an objects queue. The handling of an event from an object's queue results in an operation. O-charts also specify the creation and destruction actions for objects. The O-chart language is implemented in a tool called O-MATE with a target implementation in C++. Event specification in the Statecharts are C++-style statements.

3.4.2 Real-Time Object-Oriented Modeling (ROOM)

ROOM [Kras 95][Seli 92] is a modeling and design methodology based on the concept of actors. Actors are encapsulated, concurrent objects that communicate through directional messages sent to ports. Actors send messages to each other through channels, called bindings. Two actors can communicate only if there is a binding between them. The behavior of actors is specified using Harel's statecharts.

ROOM uses the concept of executable models as the mechanism for evolving the high level model to the design and implementation. The executable model provides the mapping from one refinement of the model to the next and eventually becomes the system implementation. The top level ROOM model conveys the fundamental hardware components and the interfaces between them. Mechanisms are provided for explicit modeling of fault tolerant mechanisms such as primary and backup versions of software. However, there is no way to specify time, a key element of real-time system modeling.

3.4.3 Octopus

The Octopus methodology [Awad 96] is a complete life-cycle methodology aimed at developing object-oriented embedded real-time systems. The Octopus approach has distinct phases:

- system requirements phase for defining and organizing the requirements,
- system architecture phase for decomposing the system into subsystems that can be developed in parallel,

- subsystem analysis phase for modeling the subsystems,
- subsystem design phase for mapping objects to explicit concurrent processes, and
- performance analysis phase for defining priorities to satisfy timing constraints.

In the system requirements phase, Octopus organizes the system requirements into use cases which relate requirements to one another. Use case diagrams, use case sheets, and system scenarios define the system requirements. The use cases cover the functions visible to the user (external functions) as well as internal functions of the system not visible to the user such as continuous monitoring of real-time conditions. A system context diagram is built from information in the use cases. The context diagram is a top level structural view of the problem domain. It shows the relationships between the system and the actors (external requesters of the system services).

In the system architecture phase, the system is decomposed into subsystems based on domains. These subsystems can be developed in parallel. The subsystem decomposition of an embedded system includes a special subsystem called the hardware wrapper. The hardware wrapper provides services for the application subsystems and isolates them from the actual hardware.

In the subsystem analysis phase, the subsystems are modeled in three views similar to the modeling views of OMT. An object model defines the static structure of the system. The object model is depicted with a class diagram showing the objects of the domain, the relationships between them, and the key attributes of the objects. Class description tables are provided for recording descriptions of each class. A functional model is provided via operational sheets that describes the services provided by the subsystems. The dynamic model is defined in a series of steps. The first step is an analysis of events. An event list is created that identifies all events and their meaning. The complexity of large numbers of events is managed by grouping events into class diagram notations for super events and sub events. Event sheets are used to record information about each event. The next dynamic modeling step is the analysis of states. Harel's statechart notations are used to model the state-dependent behavior of the concurrent subsystems.

To reduce the complexity of the statecharts, separate action tables are created that list the actions triggered by transitions, activities ongoing in a state, and events not changing the state. In further dynamic modeling steps, event significance is defined as critical, essential, ignored, or neutral. Event significance tables are used to determine the significance of events in compound states.

In the subsystem design phase, objects are mapped to processes or threads based on the analysis models. The threads are combined and expanded until they become event threads. The communication mechanism between event threads is defined. Asynchronous communication (messages) and synchronous communication (member function calls) are defined. The concurrency is designed by grouping objects based on the interaction between threads.

Prior to implementation, the performance analysis phase derives process priorities for best satisfying the required system behavior. This includes verification of the timing and concurrency behavior for the worst event sequence cases based on process priorities and timing estimations. Timing constraints are specified in the use case sheets (system requirements phase), in the event sheets, event scenarios, and event significance tables (subsystem analysis phase).

4. Passenger Train System (PTS) Requirements Specification

This section provides the requirements specification elements for an example problem that is used to illustrate the proposed methodology. The Passenger Train System (PTS) example is hypothetical and was invented for this thesis. However, it is intended to be characteristic of a realistic modeling and design problem for real-time, fault-tolerant systems.

The requirements specification contains multiple elements:

- a clear text description of the problem with illustrations to clarify the problem statement,
- short hand naming conventions for ease in understanding the specification and models
- a more formal property-based specification of critical real-time system constraints, and
- use cases for the describing the way the system operates.

4.1 Problem Statement

An amusement park has a passenger train system that allows passengers to travel from one section of the park to another. Several trains can operate at one time. Each train is short but has multiple cars. The trains all run in the same direction on a track consisting of one large loop. Passengers embark and disembark the trains at any of several stations along the track. All trains stop at all stations, regardless of whether or not there are passengers waiting to board or not. There is one siding that leads to a storage and maintenance depot. Trains requiring maintenance can be switched to this siding and removed from the operational passenger train system. Similarly, trains can enter the main track or be returned to operation by switching them back onto the main track from this siding.

The trains are powered by electric current from a power rail carried in the tracks. Power to the power rail is managed by a power control system. Each 500 yard section of track can be powered on and off independently by the power control system. Manual override can also be performed by a power station operator. When no power is applied to a section of track, a train running in that section cannot accelerate or maintain its speed. As a fail-safe mechanism, a train's braking system requires power to be disengaged. This ensures that when a train loses power, the brakes automatically engage and the train comes to a fast and smooth stop.

The automatic movement and speed of each train is controlled remotely via transmitted communication signals. Each train is equipped with a receiver and an automatic movement control system. The movement control system translates signals such as "brake", "accelerate", and "decelerate" into signals to the train's motor and brakes. Each train is also equipped with manual override controls to move and stop the train. When the manual override is in use, the automatic control is disabled.

The position of each train is monitored with sensors. Each track section contains entry and an exit sensors that detect when a train enters and leaves a section of track. In addition to detecting the presence of a train, the entry and exit sensors scan a train's unique identifier (train ID). This identifier for the front of the train is located on the right side of the front of the first car. The identifier for the back of the train is located on the left side of the back of the last car. The train ID is used by the movement control system to track the location of each train and to transmit movement control signals to a particular train.

The speed of each train is continually monitored and managed by a speed controller. Many speed sensors are placed throughout the main track. These speed sensors are placed relatively close together and cover the entire operational track sections. The speed sensors are used to detect the passing of a train at a particular point in time to determine its current traveling speed. The exact position of each speed sensor is known to the speed controller so that the distance between speed sensors can be used to compute the train's speed. The speed control system monitors the speed of trains and maintains safe operating speeds according to the maximum speed limits at the train's position in the track system.

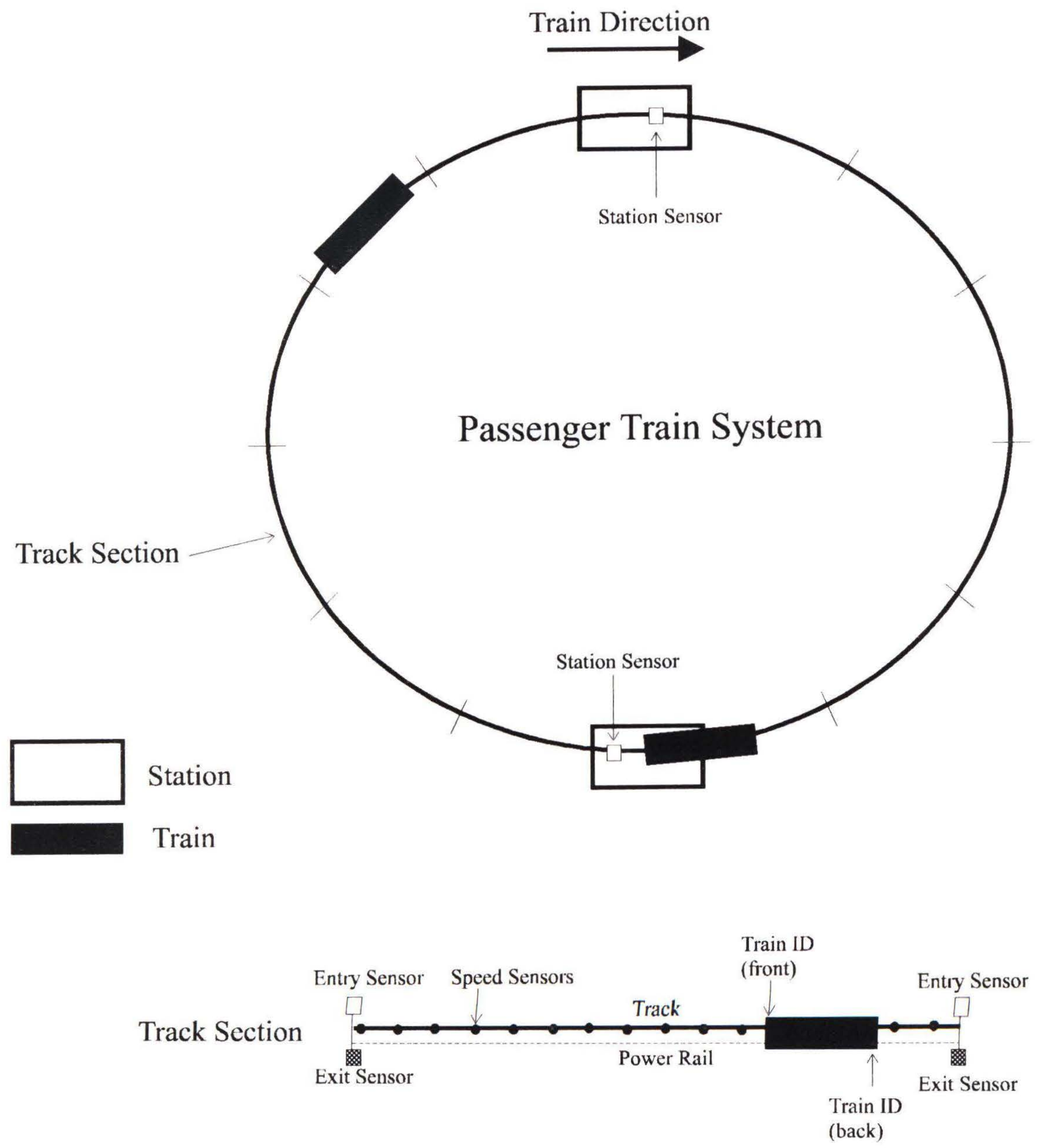


Figure 1. Passenger Train System (PTS) Configuration

A special station sensor exists in each station at precisely the point at which the train is expected to brake for a station stop. Once stopped in a station, a station operator must manually restart the train upon successful disembarking and embarking of passengers.

The most important design issue is safety of operation. System failures must result in safe actions. Trains must never be allowed to run into each other. Trains must maintain a safe distance between each other of at least 1000 yards (two track sections). Trains must never exceed a speed of 45 MPH and should normally travel at a safe speed of 40 MPH. Trains within 500 yards of a station must not exceed a safe station approach speed of 20 MPH. A traffic controller manages the position of trains. Trains can be automatically slowed, sped up, stopped, and restarted to handle the traffic flow.

In addition to being safe, the train system should be convenient for passengers. Trains should depart within 10 minutes of arrival at a station. A station stop exceeding this time results in warning messages to the station operator and automatic slowing of any train approaching the station. Trains should travel as quickly as possible (within the safety parameters) between stations to maintain passenger satisfaction with the train system. When delays are required due to traffic ahead, reducing the speed of the train is generally more acceptable for passengers than running at full speed and then stopping and waiting.

4.2 Problem Focus

The system development example in this thesis focuses primarily on the major control functions of the Passenger Train System (PTS). Particular emphasis is placed on the modeling of the interaction and behavior of the traffic and speed controllers and their lower level components. To ensure focus on the key controllers, the following simplifying assumptions have been made:

- The interaction between the power controller and the other controllers of the PTS are included in the design. However, the details of the underlying power station and power system feeding the power rails are not the focus of the design.

- The switches and track sections that are part of the storage and maintenance depot are included in the problem description to indicate that a variable number of trains can be running in the PTS at one time. The siding elements are not included in the PTS design. The focus of the design is on the control of trains within the main operational track sections.

- Interactions between the automatic controller on a train and its motor and brake systems are included in the design. However, the details of the motor and brake control systems are not included.

- The focus of this thesis is on the object-oriented model and design and its real-time, fault-tolerant characteristics. Specific algorithms for computing train speed and managing tables of data are not included. These are implementation details. General descriptions of internal operations and data is provided where necessary to understand the design.

- The physical configuration of stations and track sections is known in advance (it is part of the PTS setup program). This permits the use of attributes such as "section number", functions such as "Has Station(section number)", and constants such as "MAX SPEED" and "SAFE SPEED" to be used in the design. These terms are defined in section 4.3.

- A separate power mechanism is maintained for powering the sensors. Power to the sensors is not dependent on power in the power rail of that track section.

4.3 Short Hand Naming Conventions

Naming conventions are used throughout this design example to simplify the figures and specifications. Many of these naming conventions and definitions are used to define the safe behavior of trains traveling in different regions of the PTS track under different traffic conditions. These regions are illustrated in Figure 2. For example, when train A occupies track section 6, the condition for train B in track section 1 is "SAFE AHEAD". If train B gains on train A, and train B moves into track section 2 while track section 6 is occupied, the condition for train B in track section 2 is "TRAFFIC WARNING".

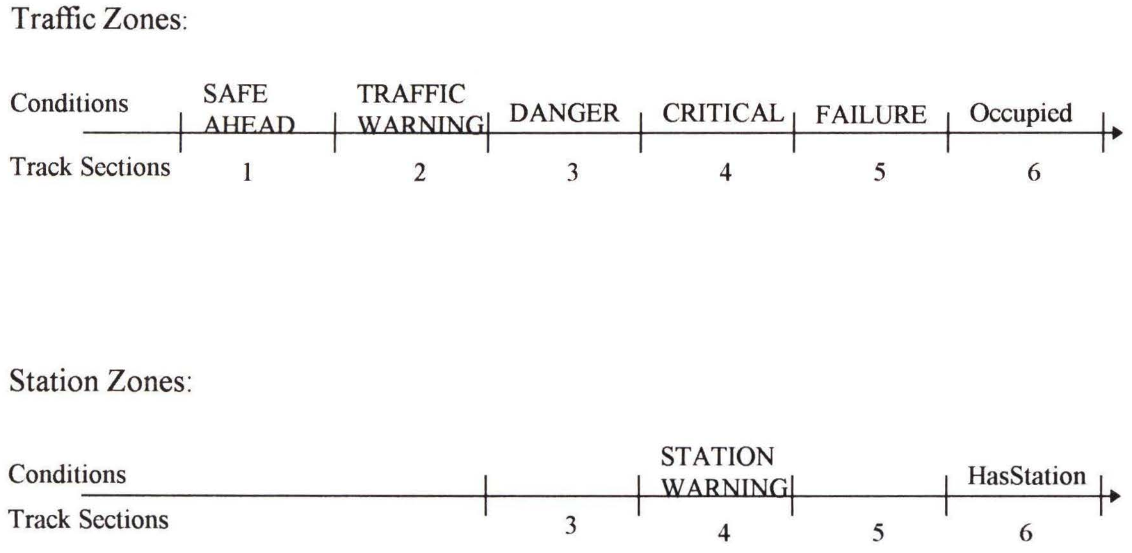


Figure 2. Illustration of Traffic Regions

Table 1 provides a list of all of the short hand naming conventions associated with the initial configuration of the PTS (track and stations) as well as the initial number of trains in the operational track sections. Table 2 provides a list of all of the naming conventions for track conditions that change depending on the position of trains. Table 3 provides naming conventions associated with a train's speed.

These speed conditions are affected by the train's position in the PTS since the top allowed speed varies depending on whether a train is near a station or not.

r	the total number of speed sensors in the PTS
s	the total number of stations in the PTS
n	the total number of operational track sections in the PTS
m	the total number of trains in the PTS
HasStation(section number)	True if there is a station in this section number
MAX STATION TIME	The maximum time a train should remain stopped in a station before departing.
WARNING TIME	The interval of time between operator warning messages when a train has been delayed in a station beyond the MAX STATION TIME.
STATION SPEED	The maximum allowable speed a train should travel when approaching a station (20 MPH in the problem description).
MAX SPEED	The maximum allowable speed a train can safely travel on any track section (45 MPH in the problem description).
SAFE SPEED	The safe speed at which a train is normally maintained to ensure that it travels below the MAX SPEED (e.g., a SAFE SPEED of 40 MPH would provide a 5 MPH threshold under the MAX SPEED).
WARNING SPEED	The speed a train should travel when the system has detected that there is either traffic or a delay ahead (WARNING SPEED < SAFE SPEED).

Table 1. Naming Conventions for PTS Configuration

HasPower(section number)	True if the power rail in this section number is powered on
Available(section number)	True if any train has entered this section number but not exited this section number.
Occupied(section number)	= NOT (Available(section number))
SAFE AHEAD	True if for all i, i=1 to 4, Available((section number + i) mod n) = True
SAFE BEHIND	True if for all i, i=1 to 4, Available((section number - i) mod n) = True
TRAFFIC WARNING	True if for all i, i=1 to 3, Available((section number + i) mod n) AND Occupied((section number + 4) mod n) = True
STATION WARNING	True if HasSation((section number + 2) mod n) = True
DANGER	True if Occupied((section number + 3) mod n) = True
CRITICAL	True if Occupied((section number + 2) mod n) = True
FAILURE	True if Occupied((section number + 1) mod n) = True
TRAIN APPROACHING	True if for some i, i=3 to 10, Occupied((section number - i) mod n) = True
WAITING TRAIN	True if a train has been slowed down as a result of a problem or delay with the current train.

Table 2. Naming Conventions for Track Conditions

SPEED OK	True if the monitored speed of the current train is less than or equal to the current allowed top speed (SAFE SPEED or STATION SPEED).
SPEEDING	True if the monitored speed of the current train is greater than the current allowed top speed (SAFE SPEED or STATION SPEED).

Table 3. Naming Conventions for Train Speed

4.4 Formal Specification

Where safety and timing issues are critical, more formal specifications are required. These formal specification make the specification more precise and easier to verify. This section provides property-based specifications for the key areas of train location, movement, spacing, and speed as well as the status of the PTS power. Similar property-based specifications are described in [Heit 95]. These specifications can be used in later development of test cases to validate the critical aspects of the design and implementation. Although the property-based specifications are more mathematical than textual, they are still fairly easy to read for non-programmers with some knowledge of boolean logic. Throughout the formal specification, the symbol "->" is used for "implies".

4.4.1 Train Location

This section specifies the exact position of a train with respect to a particular track section based upon the entry and exit sensor detection.

<u>The front of a train enters a track section</u>

For some values of i and j,

Entered(PTi,TSj) = True -> the entry sensor in track section TSj has detected that passenger train PTi has entered track section TSj but the entry sensor in TSj+1 has not yet detected passenger train PTi.

<u>The back of a train exits a track section</u>

For some values of i and j,

Exited(PTi,TSj) = True -> the exit sensor in track section TSj has detected that passenger train PTi has exited track section Tsj.

A train is entirely within one track section

For some values of i and j,

Entered(PTi,TSj) AND NOT Exited(PTi,TSj) -> Within(PTi,TSj).

4.4.2 Power Status

This section specifies the enablement and disablement of the PTS with respect to the status of power to the power rails for each track section. It also specifies the effect that removing power from a power rail where a train is traveling will have on the motion of the train.

PTS is enabled

For all i, i=1 to n, HasPower(TSi) = True -> EnabledPTS = True.

PTS is disabled

DiabledPTS = True -> for all i, i=1 to n, HasPower(TSi) = False.

Removing power from a track section stops a train that has entered the section

For some time t, and some train PTi in track section TSj, i=1 to m and j=1 to n,

HasPower(TSj) = False -> at time t+d, Speed(PTi) = 0, where d is the time it takes to stop the train when its brakes are fully applied.

4.4.3 Train Movement

This section specifies the direction of movement of trains.

Trains travel in the same direction on the track

For some values of i and j,

Within(PTi,TSj) -> NextTrack Section(PTi) = Ts(j+1) mod n.

4.4.4 Train Spacing

This section specifies the minimum safe distance between trains.

Trains always have two track sections between them

For all i, i=1 to n,

Occupied(TSi) -> Available(TS(i-1) mod n) AND Availble(TS(i-2) mod n).

4.4.5 Train Speed

This section specifies the safe operating speed of trains.

Trains do not exceed the maximum speed

For all values of time t, and all values of i, i=1 to m,

Speed(PTi) <= MAX SPEED.

Trains approaching stations do not exceed the safe approach speed

For all values of time t, and all trains PTi in track section TSj, i=1 to m, j=1 to n,

HasStation(TS(j+1)mod n) -> Speed(PTi) <= STATION SPEED.

4.5 Use Cases

Use cases are provided similar to those defined in Octopus [Awad 96]. The use cases capture the black box scenarios for the use of the system. Each use case is goal oriented and describes the main tasks of the system. Use cases specify timing requirements where applicable. This section provides a list of sample use cases for the PTS example in Table 4.

#	Name	Description
1	Station stop	A train approaches and arrives at an empty station. When the train gets within 2 track sections of the station, the train is slowed to 20 MPH.
2	Train ahead	A train approaches another train that is moving slower or is stopped. The faster train proceeds to within 5 train sections away, then 4 sections, then 3, and then 2 sections away from the train ahead. The approaching train is stopped before it gets within 2 track sections of the slowed or stopped train.
3	Station start	A station operator starts a train that was stopped in a station with no traffic ahead. The train moves out of the station at safe operating speed.
4	Station start with traffic	A station operator starts a train that was stopped in a station with traffic ahead. The train does not leave the station.
5	Station delay	A train is delayed in a station beyond the expected 10 minutes. The station operator receives warning messages.
6	Threshold speed	A train is traveling faster than normal safe operating speed (40 MPH) but less than the maximum speed (45 MPH). The train is returned to normal safe operating speed.
7	Speeding	A train is traveling faster than the maximum speed (45 MPH). The train is returned to normal safe operating speed.
8	Runaway	A train becomes a runaway. Power is removed from the power rails, stopping the train. The PTS is disabled.

Table 4. PTS Use Cases

5. PTS Object Oriented Analysis (OOA)

This section provides an example OOA static class diagram for the PTS. The class diagram illustrates the objects from the real world problem and their relationships between each other. Multiplicity is also illustrated for objects that have identical copies (e.g. trains). The analysis diagram does not convey the behavior of the system. It also does not convey any timing constraints or fault tolerant design. Behavior and timing are introduced in the dynamic model. Fault tolerance is introduced as a design refinement. The OMT analysis model for the PTS is shown in Figure 3.

The class diagram depicts the PTS as an aggregation of three main controllers. These controllers (Power Controller, Traffic Controller, and Speed Controller) represent the three main independent control functions required to manage operational functions of the PTS.

The Power Controller manages the power aspects of the PTS. It contains the power station that provides power to the power rails. The power rails are part of the Power Controller since they are turned on and off by the Power Controller. There is a power station operator. The operator class represents the user interface for the real-world power station operator (e.g., displays and manual operations).

The Traffic Controller manages the position of trains on the track. It contains the track, consisting of multiple connected track sections and multiple stations. An association between a track section and a power rail is included to show the physical configuration of track sections carrying the power rails that power the trains. For each station there are station operators. The station operators are included since they manage the movement of trains once the train is stopped in the station. Once a train has stopped and the passengers are unloaded and loaded, the station operator must initiate the train's departure via the station operator user interface. Trains do not depart the station automatically. When a train is detained in a station (past the expected 10 minute soft deadline), warning messages are displayed on the station operator's display.

The Traffic Controller class also contains a location monitor. The location monitor manages the feedback from the entry, exit, and station sensors to determine the train locations and appropriate actions for traffic control. Associations are included to indicate that an entry sensor begins a track section and an exit sensor ends a track section.

The Speed Controller controls the physical movement of trains and manages the speed of trains to maintain safe operating conditions. As such, it accepts and acts on train movement messages from the Traffic Controller. The Speed Controller contains a speed monitor that continually receives events from the speed sensors along the track. The speed monitor uses the speed sensor data to determine the measured speed of the trains. The Speed Controller attempts to adjust the train speeds whenever speed limits or thresholds are reached.

The Speed Controller also contains a transmitter that is used to send movement signals to the trains. Each train has an automatic control system that is driven by signals received from the transmitter. The trains also have motor and brake control systems that handle the physical control for speeding up, slowing down, or stopping the motor and applying or tapping the brakes. Each train also has an attendant interface and manual controls to override the automatic system. An association between the trains and the power rails is shown to indicate that a train gets its power from the power rails.

The four types of sensors in the PTS have basic characteristics and operations in common. They inherit certain of these attributes from the generalized Sensor class.

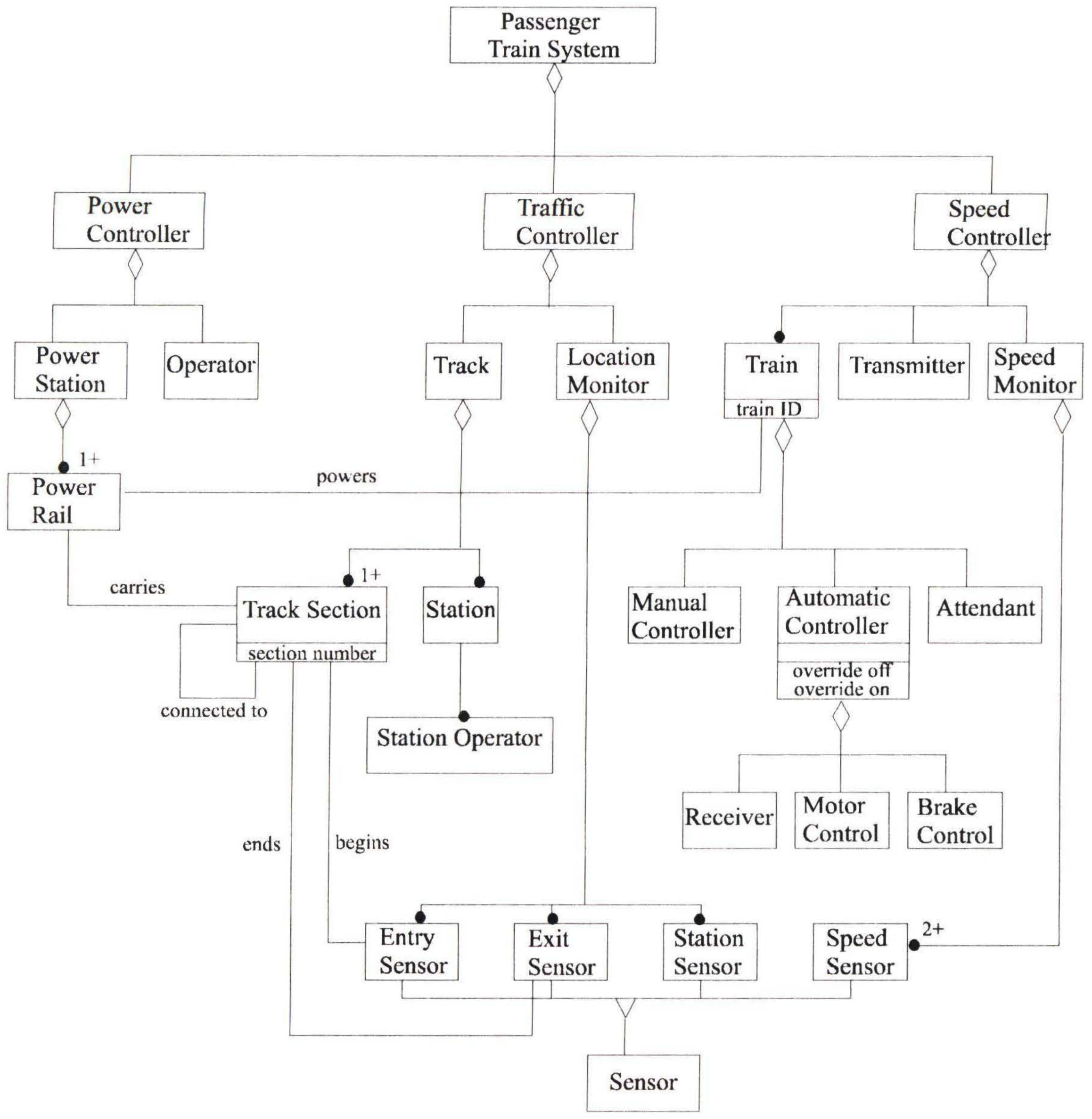


Figure 3. OOA Class Diagram

6. PTS Control System Dynamic Modeling and Design

This section provides the dynamic model and high level design refinements for selected controllers in the PTS. The high level communicating machine model is developed from the PTS problem description and class diagram. Level 2 and 3 low level state machines for two controllers illustrate the proposed dynamic modeling notation changes and introduce the timing aspects of the design. The events in the low level state machines are derived and refined using event trace diagrams for normal operational scenarios defined by use cases in the requirements specification (including normal handling of failures).

6.1 *High Level Communicating Machines (Controllers)*

In the proposed augmented methodology, the first dynamic modeling step is to create a high level model of the concurrent machines of the system. For the PTS, the major concurrent activities (running on parallel machines) are the Power Controller, Traffic Controller, and Speed Controller. Every train in the PTS also runs independently and concurrently, each with its own automatic and manual controller. The sensors are all independent external entities as well. For these high level machines, external communication is required for event notification, command communication, and synchronization of system states. The high level communicating state machine model in Figure 4 depicts the physical elements of the system. It is easily derived from the OMT class diagram.

Although it is straightforward to create the high level machine models from the problem description and the classes in the OMT analysis model, the communication channels themselves are derived through iterative modeling and design of the controller details. After the initial high level model, each major controller state machine is modeled to determine the key states, activities, and communication requirements. In some cases, each successive refinement of the more detailed state machines indicates necessary modifications to the structure of the high level machines that were not apparent in the initial model. This is demonstrated in the PTS system at the failure analysis design step where a modified high level machine model is created for fault tolerant design considerations. Care should be taken, however, not to introduce implementation-level details into the high level analysis model. Details such as object creation and destruction associated with programming languages, although required at the implementation level, only introduce complexity and confusion into the high level model.

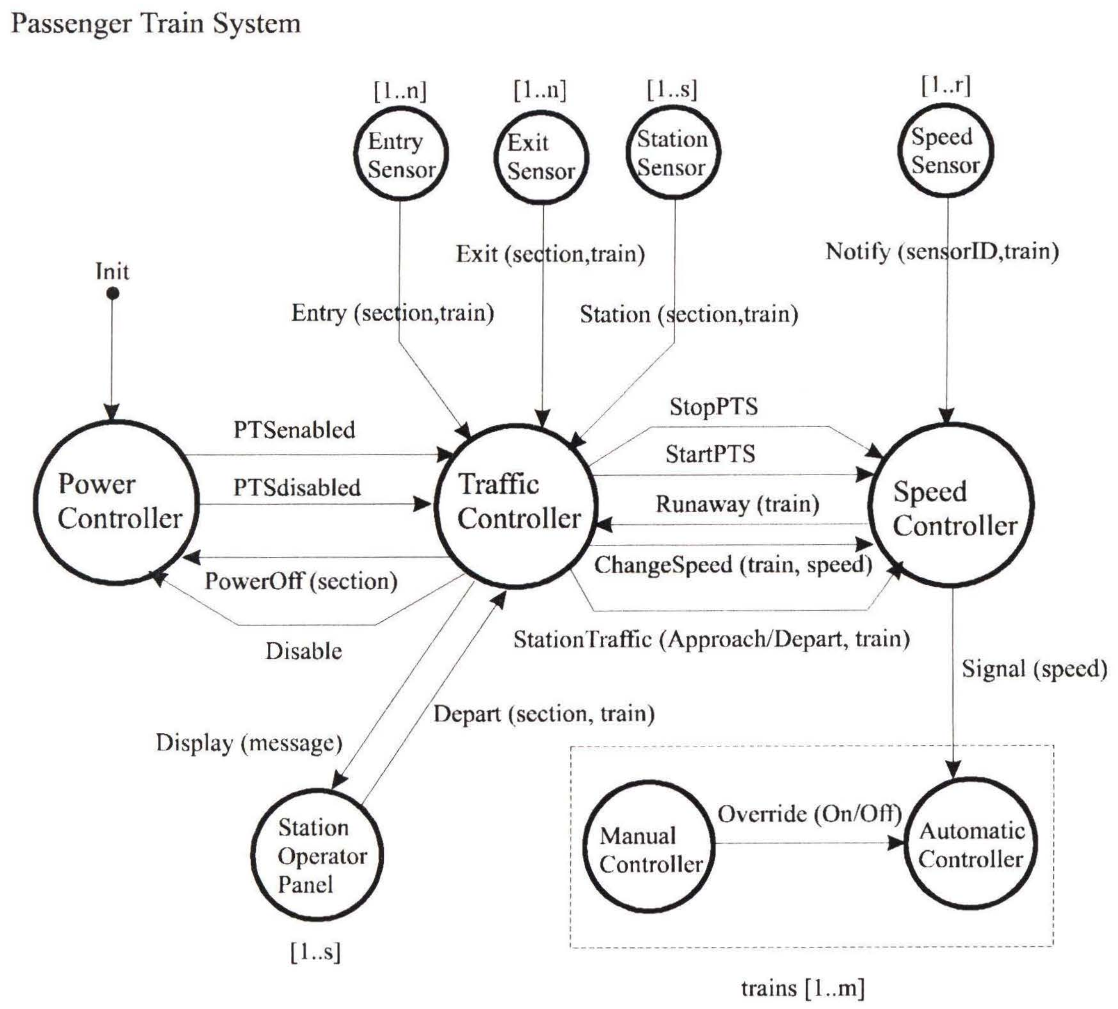


Figure 4. High Level Communicating State Machines

The first controller that starts when the PTS system is initiated is the Power Controller. Once good power status of the system has been determined, the Power Controller starts the Traffic Controller with a PTSenabled message. If at any point in the operation of the PTS, power is disabled, the Power Controller will give the Traffic Controller a PTSdisabled message.

Once started, the Traffic Controller starts the Speed Controller with the StartPTS command. Similarly, when the Traffic Controller is given the PTSdisabled message, it also stops the Speed Controller with the StopPTS command.

The Traffic Controller manages the location of trains with respect to each other and also with respect to their proximity to stations. The Traffic Controller can detect certain failure cases when a train does not appear to be responding to normal control signals. To prevent catastrophic events, the Traffic Controller can request that specific track sections be powered off (PowerOff command) or that all track sections in the PTS be powered off (Disable command). Removing the power from the power rails causes power loss to the trains and the physical fail-safe braking mechanism stops the trains.

There are multiple independent machines modeled for each of the many entry, exit, and station sensors. These machines are all very similar in that they continually scan passing trains, reading the train ID when a train passes. Upon detection, each sensor notifies the Traffic Controller of the event by sending the appropriate event notification message and event data. For all entry, exit, and station events, the Traffic Controller receives two pieces of information; the section number identifying the particular sensor location and the train ID of the passing train.

The Traffic Controller communicates with the Speed Controller to request that the speed of trains be changed (ChangeSpeed message). The Traffic Controller issues these commands to control the flow of traffic in the PTS. For example, when a train is approaching a station, the Traffic Controller requests that the train's speed be reduced to the station approach speed (STATION SPEED) of 20 MPH. When a train departs a station, the Traffic Controller requests that the train's speed be increased to the normal safe speed (SAFE SPEED). The Traffic Controller monitors the position of trains and issues requests to the Speed Controller to adjust the speed (and therefore position) of trains. When the Traffic Controller needs to stop a train under normal operating conditions, it issues a ChangeSpeed message with a speed of zero. The desired effect is that the train brakes smoothly to a stop.

Through the speed sensors, the Speed Controller monitors the externally measured speed of trains and sends speed control signals to trains. There are many speed sensors in the PTS. These speed sensors are very similar in operation to the other sensors in that they scan for the passing of trains and report the event and train ID to the Speed Controller. Since there are many more speed sensors than track sections, the speed sensors must each have a unique identifier that identifies their location to the Speed Controller. Using the speed sensor data, the Speed Controller can detect when a train may be traveling beyond the safety speeds. When all normal attempts to slow a speeding train have failed, the Speed Controller will give a Runaway train message to the Traffic Controller. Upon receiving the runaway notification, the Traffic Controller can take preventive actions such as powering off the appropriate track section or disabling (powering off) the entire PTS to avoid a catastrophe.

The Traffic Controller also notifies the Speed Controller when a particular train is approaching or departing a station (StationTraffic message). The Speed Controller uses this notification to set the appropriate top speed for the train.

The station operator panels in each station are used by the station operators to signal that the train is ready to depart the station. The Traffic Controller requests that appropriate status and warning messages be displayed on the station operator panel.

6.2 Communicating State Machines

The second level in the proposed hierarchy is the modeling of the dynamic functions of the main controllers using state machines. This section provides example state machines for the Traffic and Speed Controllers of the PTS. In this example, a third level state machine for a sub-state of the Traffic Controller is also illustrated. The use of superstates and their corresponding substates is consistent with the OMT methodology and is not used in Shaw's CRSMs. For the Traffic Controller, the use of the Station Event Handler sub-state makes the main controller behavior more readable. The details of the train stopping and starting within the station are left to a lower level refinement.

6.2.1 Traffic Controller State Machines

The second level Traffic Controller state machine is illustrated in Figure 5. The Traffic Event Handler is the main Traffic Control state and is entered upon notification that the PTS is enabled. The Traffic Controller starts the Speed Controller with the StartPTS external command. The Traffic Event Handler continually monitors for events until it receives a PTSdisabled external notification from the Power Controller. Possible events are:

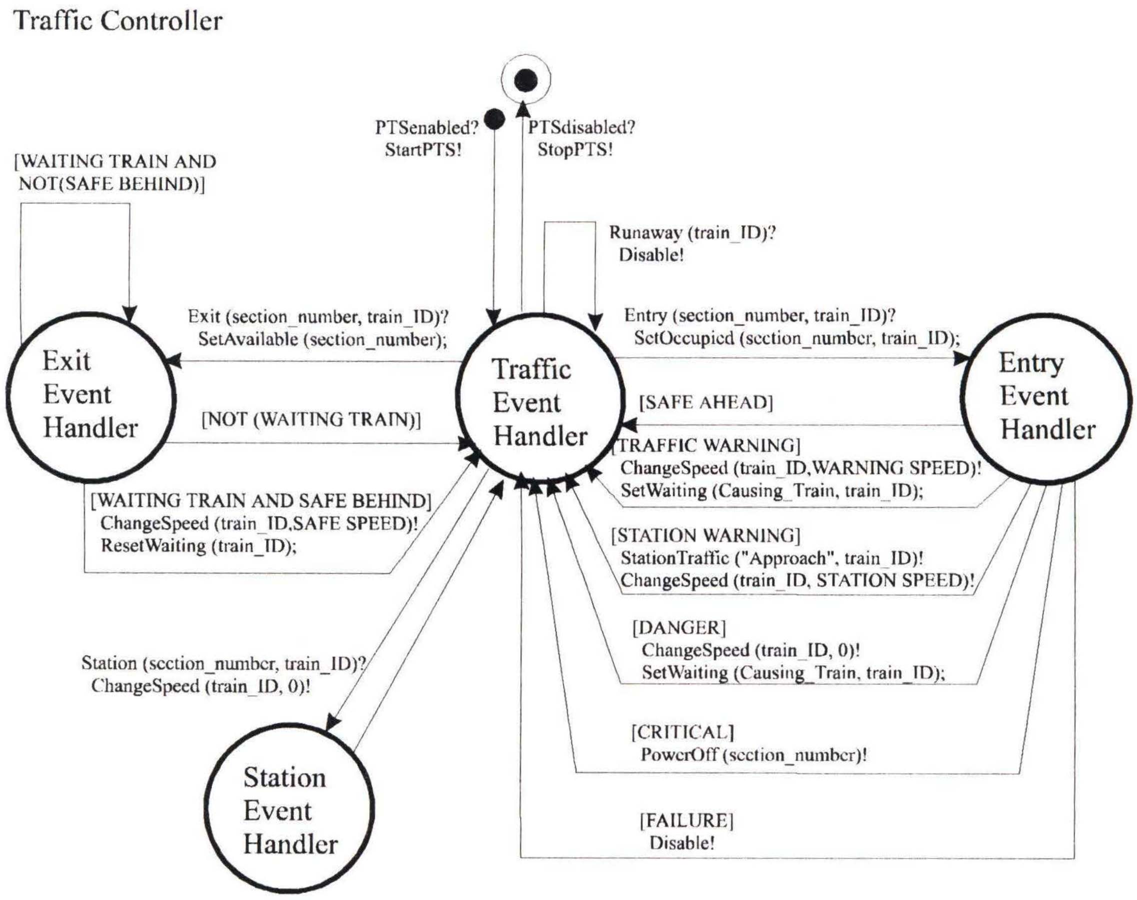
- Entry notification from an entry sensor
- Exit notification from an exit sensor
- Station notification from a station sensor
- Runaway notification from the Speed Controller
- PTS disabled notification from the Power Controller

The Traffic Controller maintains current, internal state information on the location and status of trains operating in the PTS. State is maintained for each track section indicating whether it is available or occupied. Internal commands (SetAvailable and SetOccupied) are used to set the availability status of track sections. Other internal state information managed by the Traffic Controller is the waiting status of trains. Trains must be put in waiting state if there is traffic ahead that would prevent them from maintaining a safe operating distance. The SetWaiting and ResetWaiting commands are used to preserve this state.

Upon detection of an Entry event, the Traffic Controller marks the track section as occupied and determines the current traffic condition according to the traffic regions described in Figure 2. When it is safe ahead, the train is allowed to proceed. When a train has entered a TRAFFIC WARNING zone (indicating that a track section 4 sections ahead is occupied), then the train is slowed to a warning speed. The internal SetWaiting command is executed to record the fact that the train was slowed as a result of the train ahead. When the train ahead moves to a distance away that is deemed to be safe (4 track sections between trains), the waiting train is returned to normal speed. This is done in an attempt to smooth the flow of traffic and not cause lots of starting and stopping when trains get close together. Similarly, when a train has entered a STATION WARNING zone it is slowed. The Speed Controller is notified of the station approach so that the top speed of the train can be set appropriately for speed safety checks. When a train enters a DANGER zone it is stopped to maintain the safety requirements (at least two track sections between trains). If by chance a train gets closer than the DANGER zone, then a failure has occurred somewhere. In the CRITICAL zone the specific track section is powered off to stop the train. In the FAILURE zone, the entire PTS is disabled (powered off). This is intended to prevent trains from violating the specification requirements for safe distance between trains and from running into each other under unspecified failure conditions.

Upon detection of an Exit event, the Traffic Controller marks the track section as available. It then checks behind the exiting train to see if an approaching train was slowed to avoid getting too close to the current train (WAITING TRAIN = true). If there are at least 4 track sections between the trains (SAFE BEHIND), then a message is given to the Speed Controller to return the train to the normal safe operating speed. The approaching train is then marked as no longer waiting.

Upon detection of a Station event, a message is given to the Speed Controller to stop the train. The Station Event Handler is a super-state for the events that take place in the station from the time the train is stopped to the time the train departs.



Traffic Controller
PTSenabled?
StartPTS!
PTSdisabled?
StopPTS!
[WAITING TRAIN AND NOT(SAFE BEHIND)]
Runaway (train_ID)?
Disable!
Exit (section_number, train_ID)?
SetAvailable (section_number);
Entry (section_number, train_ID)?
SetOccupied (section_number, train_ID);
Exit Event Handler
Traffic Event Handler
Entry Event Handler
[SAFE AHEAD]
[NOT (WAITING TRAIN)]
[TRAFFIC WARNING]
ChangeSpeed (train_ID,WARNING SPEED)!
SetWaiting (Causing_Train, train_ID);
[WAITING TRAIN AND SAFE BEHIND]
ChangeSpeed (train_ID,SAFE SPEED)!
ResetWaiting (train_ID);
[STATION WARNING]
StationTraffic ("Approach", train_ID)!
ChangeSpeed (train_ID, STATION SPEED)!
[DANGER]
ChangeSpeed (train_ID, 0)!
SetWaiting (Causing_Train, train_ID);
Station (section_number, train_ID)?
ChangeSpeed (train_ID, 0)!
[CRITICAL]
PowerOff (section_number)!
Station Event Handler
[FAILURE]
Disable!

Figure 5. Traffic Controller State Machine

Figure 6 illustrate the Station Event Handler state machine. This level-three state machine satisfies the problem requirements for timely departure from stations with warnings to station operators when time limits are exceeded. The station operator is the only one who can initiate a train's departure from a station. The Station Event Handler sub-state of the Traffic Controller can delay the train from departing due to traffic conditions ahead.

Upon entry into the Station Event Handler, a timer is set to give an indication to the station operator that a train is late if the station time exceeds the MAX STATION TIME (10 minutes). If the train continues to be delayed after the warning, periodic warning messages are provided in intervals equal to WARNING TIME. Additionally, if a train is approaching the station (within 10 track sections of the station), the approaching train is slowed and marked as waiting on the train that is delayed in the station. When the passengers have been loaded, the station operator initiates the train's departure via the operator's console. This results in a Depart message to the Traffic Controller. If the departing train is in a SAFE AHEAD zone (no trains within 4 track sections ahead of it), then the train is allowed to depart and its speed is set back to normal safe operating speed via the Speed Controller. If it is not safe ahead, the train remains delayed in the station until the track sections become available ahead.

Station Event Handler

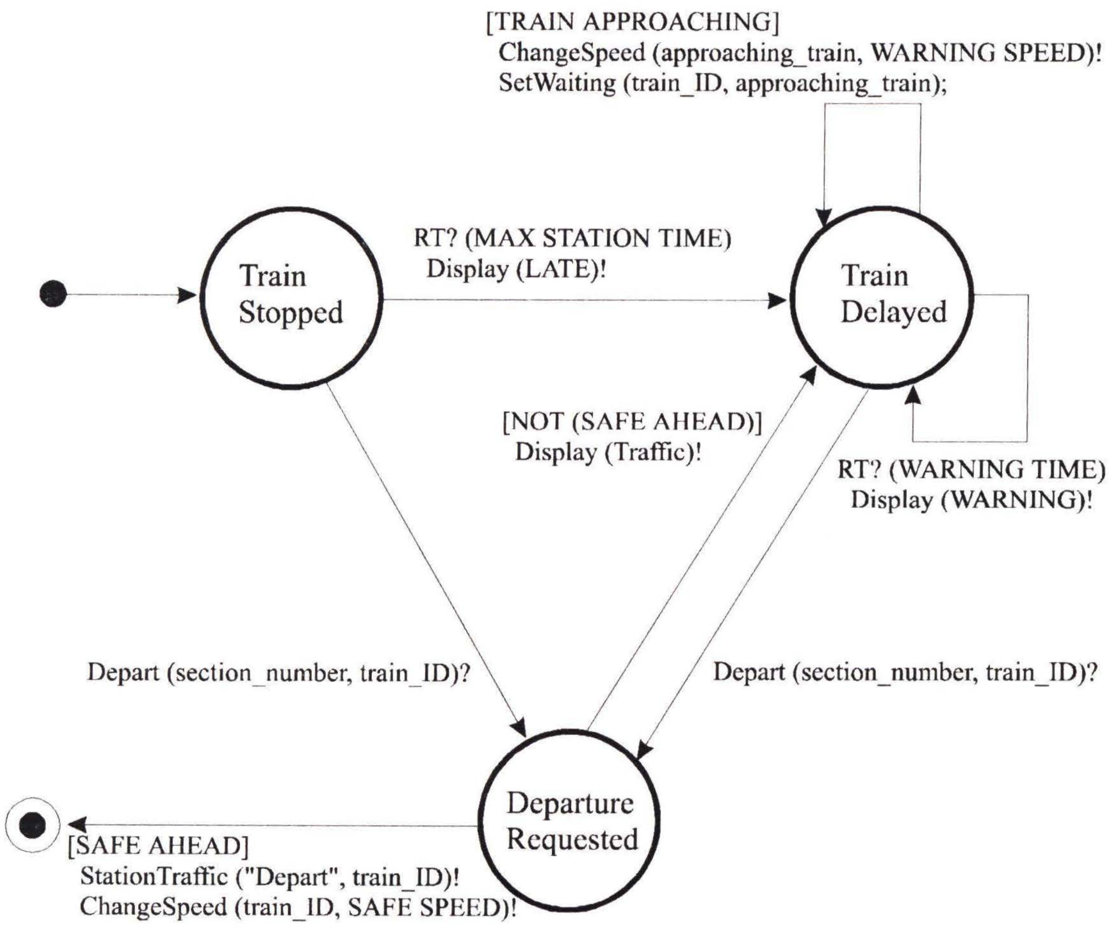

[TRAIN APPROACHING]
ChangeSpeed (approaching_train, WARNING SPEED)!
SetWaiting (train_ID, approaching_train);
RT? (MAX STATION TIME)
Display (LATE)!
Train Stopped
Train Delayed
[NOT (SAFE AHEAD)]
Display (Traffic)!
RT? (WARNING TIME)
Display (WARNING)!
Depart (section_number, train_ID)?
Depart (section_number, train_ID)?
Departure Requested
[SAFE AHEAD]
StationTraffic ("Depart", train_ID)!
ChangeSpeed (train_ID, SAFE SPEED)!

Figure 6. Station Event Handler State Machine

6.2.2 Speed Controller State Machine

The level two Speed Controller state machine is illustrated in Figure 7. The Speed Controller handles the following types of events:

- Change Speed message from the Traffic Controller to change the speed or stop a train
- Notify event from a speed sensor to monitor the detected speed of trains
- Station Traffic (depart or approach) message from the Traffic Controller
- Stop PTS notification from the Traffic Controller

When a request is received from the Traffic Controller to change the speed of a train, the Speed Controller signals the train (via the transmitter) of the requested new speed as long as the speed is different from the current measured speed. Requests to stop trains are performed with a ChangeSpeed command containing a zero speed. The Speed Controller maintains internal data on the measured speed of trains.

When the Speed Control Handler receives a Station Traffic depart or approach message it sets the train's top speed to the maximum appropriate safe speed. For example, when approaching a station, the maximum safe speed is 20 MPH, so the top speed for that train is now 20 MPH (STATION SPEED). After departing the station, the maximum safe speed is 40 MPH so the top speed is set back to 40 MPH (SAFE SPEED).

The Speed Control Handler continually monitors the speed of trains by receiving speed sensor notifications as trains pass the speed sensors. Since there are lots of speed sensors, this activity represents the majority of the workload for the Speed Controller. Upon receiving new speed sensor information, the new measured speed of the train is determined and saved (ComputeSpeed). If the train's speed is less than or equal to the top speed (SPEED OK) then no action is taken. In either case, a try counter is reset. This try counter is used when a train is speeding to count the number of attempts to slow the train. At the first try, the train is signaled to operate at its top speed. At the second try, the train is signaled to stop since the previous attempt to slow the train back to its top speed presumably failed. At the third try, it is assumed that there has been a major failure and that the train is out of control. The runaway train message is given to the Traffic Controller.

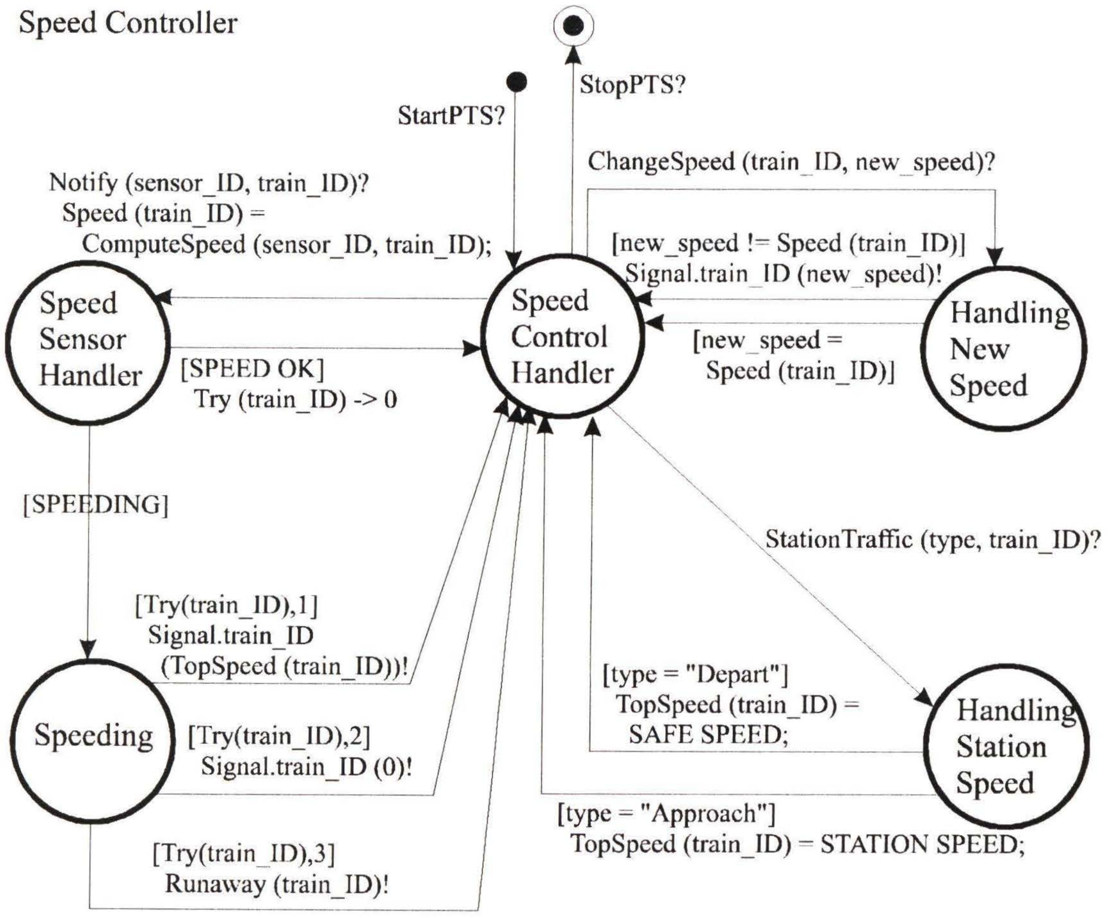


Figure 7. Speed Controller State Machine

6.3 *Event Trace Diagrams*

The details of the state transitions (messages) of the controllers are partially derived through event trace diagram scenarios. This section provides example normal operational scenarios taken from use cases for the key operations and interactions between the Traffic and Speed Controllers. The event trace diagrams are not generalized or comprehensive illustrations of the controller operations. They are specific scenarios that could occur in a time sequence. The three example scenarios were chosen because they represent most of the interesting functions of the controllers. Each instance of an object is represented by a vertical line. For simplicity of the diagram, sensor objects are combined into one line.

6.3.1 Train Ahead (Use case 2)

Train PT1 has exited track section TS9 and entered track section TS10. PT1 is traveling very slow or has been delayed at a station stop in TS10 (either condition results in the same operational flow with respect to the approaching train PT2). Train PT2 has entered track section TS5 and is traveling at a safe operating speed of 40 MPH.

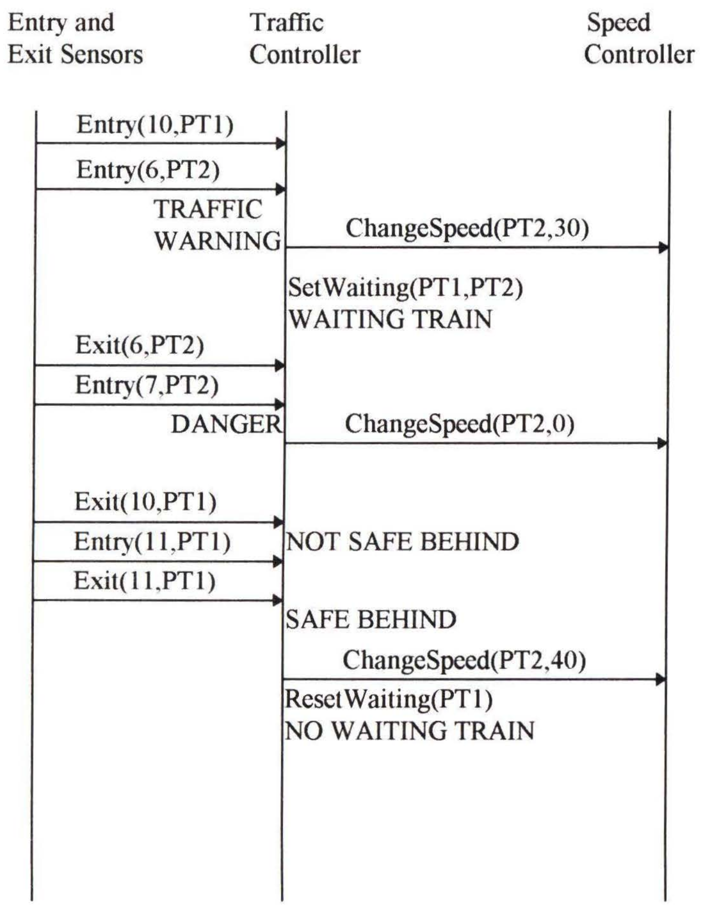


Figure 8. Event Trace Diagram for Use Case 2

6.3.2 Station Stop (Use case 1)

There is a station at track section TS10. Train PT1 has exited track section TS6 and entered track section TS7 traveling at a safe speed of 40 MPH.

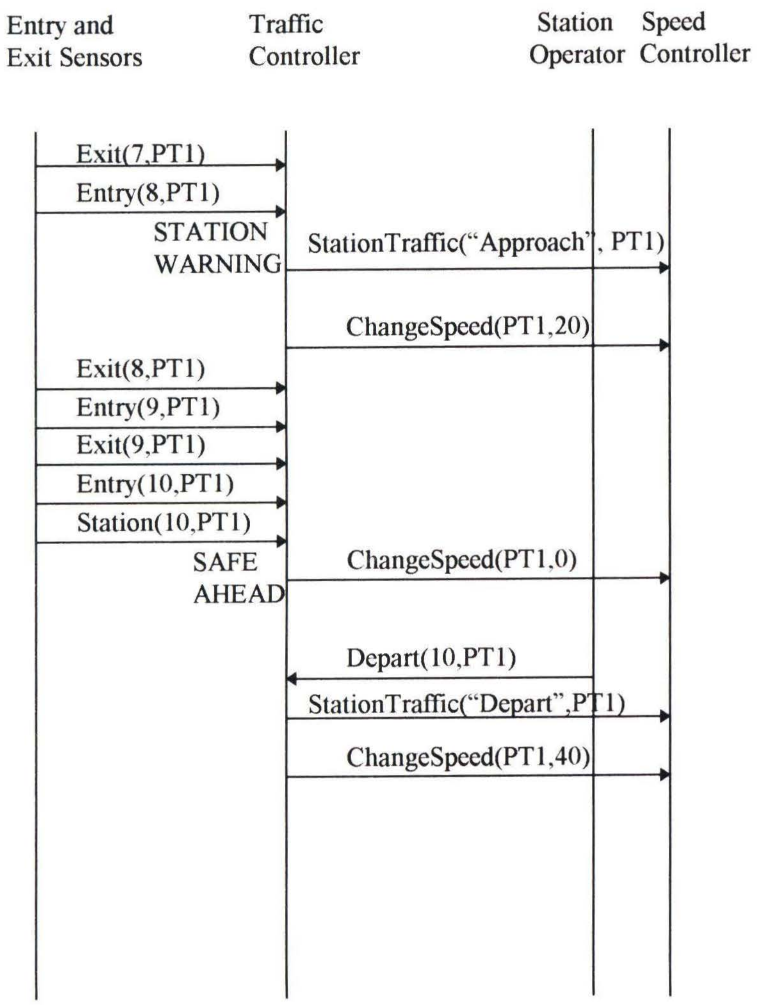


Figure 9. Event Trace Diagram for Use Case 1

6.3.3 Runaway (Use case 8)

Train PT1 has had a failure of some sort that affects the train's motor control and speed. The failure is causing PT1 to accelerate beyond the safe speed of 40 MPH. The speed sensors (S1-Sx) detect that PT1 is accelerating and take corrective action.

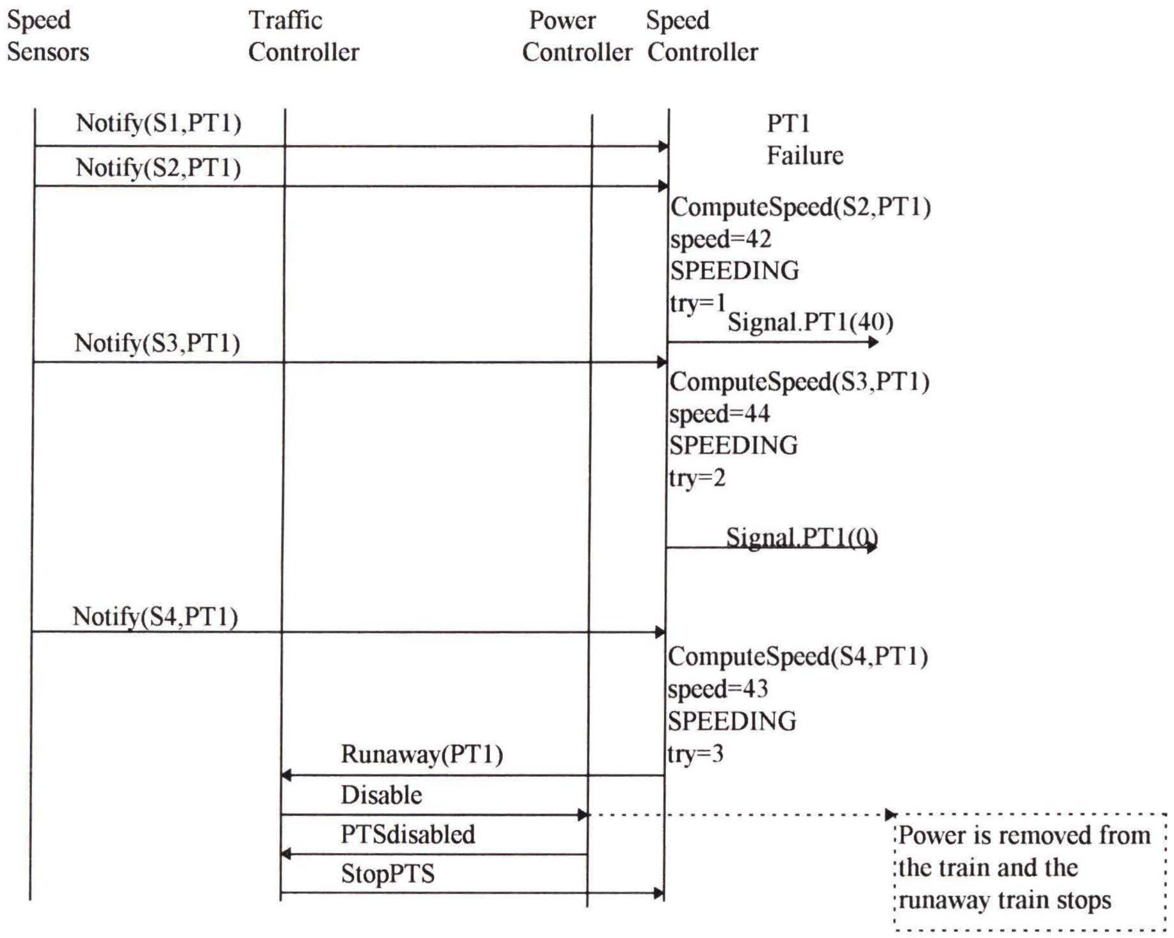


Figure 10. Event Trace Diagram for Use Case 8

7. PTS Fault Tolerant Design

Although error handling and safety were of considerable concern in the modeling and design of the PTS, the design is not fault tolerant. Since passenger safety is a primary issue in the PTS, performing fault analysis is an important step in the design process. The PTS hardware and software failure scenarios must be considered and the appropriate design changes incorporated into the successive refinements of the model. The initial OMT OOA model remains unchanged, since it is intended to reflect the statement of the problem requirements, not the details of the design.

7.1 *Component Failure Scenarios*

This section provides an example analysis of possible critical hardware and software component failures that could occur in the PTS. The ability to detect the failure, the severity of the failure's effect, the repair/down-time, and the safety vs. cost tradeoffs are considered in the analysis. For simplicity, single component failures are primarily considered.

7.1.1 Entry Sensor Failure

If an entry sensor fails to detect the entry of a train into a track section, the results could be catastrophic. With the existing design, a track section is determined to be occupied based on the data provided by the entry sensors.

Severity

If a train exits one section and enters another and the entry sensor fails in the new track section, the Traffic Controller does not know that the new track section is occupied. If the train continues to move forward and reaches the exit sensor, the failure can be detected. However, if the train slows or stops in that track section before reaching the exit sensor, there is a window of opportunity for the Traffic Controller to allow another train to run into the back of the slowed or stopped train since the track section will be assumed to be available. The Traffic Controller will incorrectly determine that it is SAFE AHEAD when it is not, with possibly catastrophic results. In this case, extra cost is warranted to mask and detect the failure.

Although less likely, an entry sensor could also fail by reporting that a train entered a track section when it actually did not. The track section would be erroneously marked as occupied and traffic delays could occur. The normal operation of the PTS could be affected, but no catastrophic events would be allowed to happen.

Solution

Each track section contains three entry sensors in a Triple Modular Redundancy (TMR) configuration, instead of one. Each of the three sensors detects the presence of a train and a separate voter compares and votes on the result. If one sensor fails, it is detected and masked. The trains could remain in safe operation and the PTS operators alerted to the failure. The necessary repairs could be made quickly since the exact failure can be pinpointed.

With this TMR solution, when one sensor has been determined to be faulty, the system could be reconfigured to use the remaining two sensors. Comparison of the two remaining operational sensors detects if a second subsequent failure occurs. When a second failure is detected the PTS could be completely shut down for repair. This second failure would take longer to isolate since the comparator cannot determine which input is incorrect, only that there is a mismatch.

Depending on the cost of the sensors, this solution could add significant cost to the PTS. Each sensor must be triplicated, a voter and comparator must be provided, and control logic must exist for monitoring the multiple inputs to determine any appropriate actions. In the most fault tolerant solution, the voter should reside at the location of the Traffic Controller, not at the tracks with the sensors. This is to ensure that failures in the physical lines connecting the sensors to the controller input are detected as well as failures in the sensors themselves. The cost and complexity would be significant, but appears to be warranted by the severity of the failure.

As an alternative to TMR, a less rigorous and less costly solution would be to mirror (duplicate) each sensor and provide a comparator to determine if there is a failure in one of the sensors. If both sensors fail in the same way, the error will not be detected. When a single failure occurs, the PTS must be shutdown for repair. The repair time is longer than with TMR since the specific failing sensor cannot be identified.

7.1.2 Exit Sensor Failure

With the existing design, a track section is determined to be available based on the data provided by the exit sensors. This means that if an exit sensor fails to detect a train, then a track section will remain in an occupied state when it is no longer occupied.

Severity

When the exit sensor fails to detect a train, the failure could be detected at the next exit sensor. The effect of the failure would be minimal in that an approaching train might be slowed or stopped unnecessarily but no catastrophic failure would occur.

A more serious consequence would occur if an exit sensor failed by reporting that a train has exited a track section when it has not. This type of failure would be less likely to occur. However, since the PTS design uses the exit sensor to indicate that a track section is available, an undetected erroneous report of a train exit could result in an occupied track section being marked as available, with potentially catastrophic results.

It is much more likely that an exit sensor would fail to detect a train than to detect a train erroneously. In the former case, the effect of the failure is minor and does not provide a safely hazard. However, it should be detected and repaired to fix the resulting traffic delays.

If by chance an exit sensor erroneously reports a train's exit, there will likely be other indicators that can be used to detect the problem. These include incorrect parameter information on the sensor data and out of sequence exit and/or entry notifications for the train identified.

Solution

Since there is a possibility (even if remote) that a catastrophic event could occur, some cost appears to be warranted to detect the failure. Mirrored exit sensors with a comparator function could prove to be a cost effective solution considering that safety is the number one design issue. The comparator function detects the failure, but the failure could not be isolated to a particular sensor. As a result, the repair time could increase slightly. Parameter checks and out of sequence entry/exit checks should also be added to the design.

7.1.3 Station Sensor Failure

If a station sensor fails, the trains will not stop at the station. Instead, they will continue through the station at safe station approach speed (20 MPH). Normal traffic control would be maintained via the exit and entry sensors, so no collisions would occur. However, because the train does not stop at the station, there is no mechanism for returning it to safe operating speed (40 MPH).

Severity

Detection of the failure would be obvious since the train would not stop at the station. Therefore, the problem could be quickly isolated and corrected with only minor inconvenience to passengers. The station containing the sensor would need to be temporarily closed to passenger loading and unloading.

With the current PTS design, a station sensor failure would eventually result in traffic delays unless a mechanism is introduced to return the train to normal operating speed once it passes the station. The current speed return mechanism is initiated by the station operator manually causing a train to depart. This causes the Speed Controller to return the train's top speed value to the normal safe operating speed. Without this mechanism, trains passing through the failing station will remain at station speed until they reach and depart from the next station on the line.

Solution

The use of the exit sensor in a track section containing a station could be redefined to have an additional action that is to return the train to normal safe operating speeds. This eliminates the traffic delays caused by the failure.

It may be unacceptable to have a station temporarily closed for repair of the failing station sensor. In this case, replicated station sensors may provide a better solution. One sensor is in operation, the other is a spare. This provides a fast method of keeping the station operating. If the operator observes that a train does not stop at a station, the failing sensor could be temporarily disconnected, the spare sensor connected, and the station could remain in operation until a convenient repair time. Traffic delays occur temporarily but the PTS could be returned to normal operation very quickly. The use of the spare sensor allows the station operator to completely mask the effects of the failure once it has been detected.

7.1.4 Traffic Controller Software Failure

There are two categories of failure affecting the operation of the Traffic Controller software: a machine failure and a software failure. If the machine hardware fails that is running the Traffic Controller software, then the Traffic Controller software cannot operate. Messages (including sensor inputs and runaway notifications) will not be received and acted upon.

A software failure could be less predictable. In addition to the possibility of ignoring or failing to act upon messages, a software failure might result in undetected erroneous messages and actions.

Severity

Any failure in the Traffic Controller, whether it be of a hardware or software nature could be catastrophic. A failure in the Traffic Controller would result in trains in motion without safety controls for slowing at stations or maintaining safe distances between trains.

Solution

To detect complete failure of the machine running the control software and for software failures such as endless looping, a Watchdog Controller could be used. The Watchdog Controller runs on separate hardware from the Traffic Controller and regularly polls the Traffic Controller to determine it is operational. Erroneous results from the Traffic Controller are not detected, but the Watchdog Controller detects complete failures. Upon detection of a complete failure, the watchdog would disable the power to the PTS system via the Power Controller to prevent speeding trains and collisions.

To help detect software failures, multiple versions of the Traffic Controller software could be utilized. These different software versions would be implemented by different developers. One version would be the primary operational version. The other version would receive a copy of all inputs and would compare the outputs from the first version with its own computed outputs. Any discrepancies would result in an immediate notification to the PTS watchdog for shutdown or recovery actions. This is a very costly solution due to the added development expense for developing, testing, and maintaining multiple versions of the software and synchronizing the output comparisons. However, in this type of application, the cost could be warranted. Another negative result of this solution is the complexity it adds to the Traffic Controller design and implementation. The design of the watchdog recovery and shutdown could be complex depending on the degree of sophistication desired. Another potential problem to be addressed with multiple versions is the timing overhead of the additional communication to the second software version and the delay in determining if a failure has occurred.

A lower cost tradeoff solution would be to design and develop a single version of the Traffic Controller with stringent design methodology, development process, and verification steps. This does not guarantee the elimination of software failures, but would result in fewer software defects and better detection and recovery of defects that might occur.

7.1.5 Speed Sensor Failure

The speed sensors are used by the Speed Controller to measure the actual speed of trains traveling in the PTS. If a speed sensor fails to report the detection of a train, the next speed sensor will detect the train and the computation of the train's speed will be delayed. Since speed sensors have unique identifiers and have known positions on the track, the Speed Controller can still accurately compute the speed of the passing train.

Severity

If speed sensors are located far apart, a failure of one speed sensor to detect a train could result in an unacceptable delay in determining that a train is traveling beyond the safe speed limits or is traveling out of control. However, when a speed sensor does fail, it would be easily detected and repaired since trains travel in one direction on the tracks and the sensor ids could be used to determine that a gap exists in the sensor notification.

Solution

The speed sensors should be placed close enough together so that there is a short amount of time between new computations of the train's speed, even when a sensor fails and there is a gap in the sensor notification. This raises the cost of the PTS by the cost of the extra sensors. It also increases the communication and processing overhead of the Speed Controller in handling the speed sensor notifications.

7.1.6 Transmitter Failure

The Speed Controller uses a single PTS transmitter to send control signals to trains for the purpose of maintaining safe operating speeds and safe distances between trains. These control signals are transmitted to receivers located in each train.

Severity

If the transmitter fails completely or produces erroneous signals, it has the same effect as if the Traffic Controller and/or Speed Controller itself had failed. Trains would run unchecked by the PTS controllers. This would certainly result in catastrophic failures.

Solution

The PTS contains three transmitters in a TMR configuration, instead of one. The transmitters transmit identical signals in parallel to a train using three different frequencies. Each train receives the three signals and majority vote on the signal to use to control the train. By placing the voter in the train, failures resulting from the signal transmissions (noise, etc.) are detected as well as failures in the transmitter itself. The voter waits a very short amount of time for all three signals. Any signal not received within that time period is determined to be a failure. Failures (signal timeouts or mismatches) are reported to the train attendant as a visual and/or audible warning.

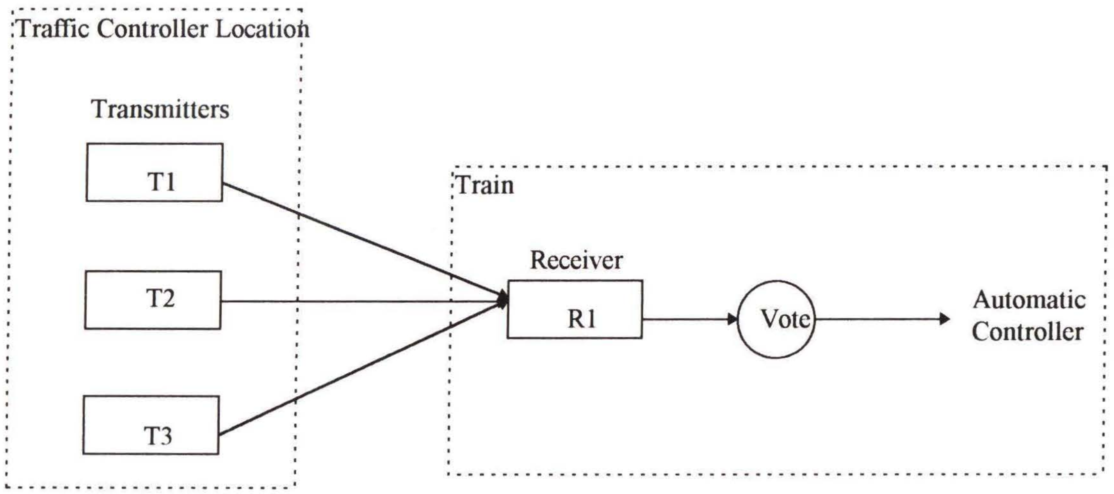


Figure 11. Triple Replicated Transmitter with Voting Receiver

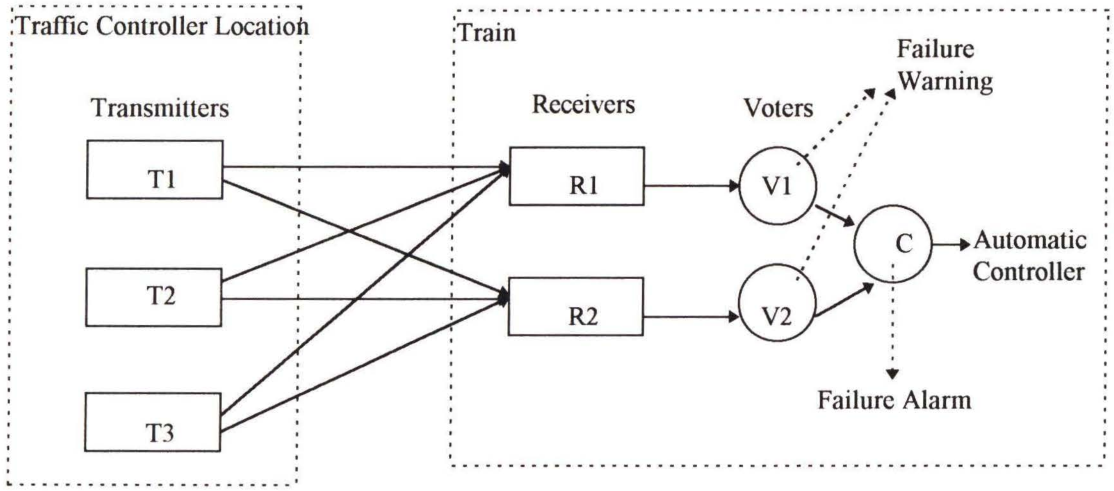


Figure 12. Mirrored Receiver with Comparator

7.1.7 Receiver Failure

There is a receiver on each train that receives the speed control signals. These signals are used to control the automatic operations of the train's motor and brakes.

Severity

If a receiver fails completely, signals could be lost and it would be the same as if the Speed Controller and/or Traffic Controller functions had failed. If a receiver receives a signal incorrectly, erroneous operations could be applied to the train's motor and brakes. In either case, catastrophic failures could occur.

Solution

Provide two (mirrored) receivers on each train. Each receiver receives the three signals from the TMR transmitters and each receiver votes on the transmitted signal. A single comparator then compares the results from the voting receivers. If the results are different, an alarm is sounded to the train attendant who initiates manual override. This solution incurs additional cost for the receivers and comparator, but the cost would be warranted by the severity of failures.

7.1.8 Speed Controller Software Failure

The Speed Controller software monitors and controls the speed and operation of trains. It is the sole interface to the automatic control functions on the trains. Similar to the Traffic Controller, the Speed Controller software can be affected by either a hardware or software failure. If the machine hardware fails that is running the Speed Controller software, then the Speed Controller cannot operate. Messages (including speed sensor inputs and change speed commands to prevent collisions) will not be received and acted upon.

A true software failure could be less predictable. In addition to the possibility of ignoring or failing to act upon messages, a software failure might result in undetected erroneous messages and actions.

Severity

Hardware or software failures affecting the Speed Controller software could produce catastrophic results since trains would be running out of control. A failure in the Speed Controller would result in trains in automatic motion without speed monitoring and without the ability to reduce a train's speed or bring it to a normal stop.

Solution

To detect complete failure of the machine running the control software and for software failures such as endless looping, a Watchdog Controller could be used. The Watchdog Controller runs on separate hardware from the Speed Controller and regularly polls to determine if the Speed Controller is operational. This would not detect erroneous results, but would detect complete failures. Upon detection of a complete failure, the watchdog disables the power to the PTS system via the Power Controller to prevent speeding trains and collisions.

As with the Traffic Controller solution, multiple versions of the Traffic Controller software could be utilized to help detect software failures. This is a very high cost solution. Rigorous design and development methodologies and exhaustive verification processes would be suggested as a lower cost tradeoff alternative.

Passenger Train System

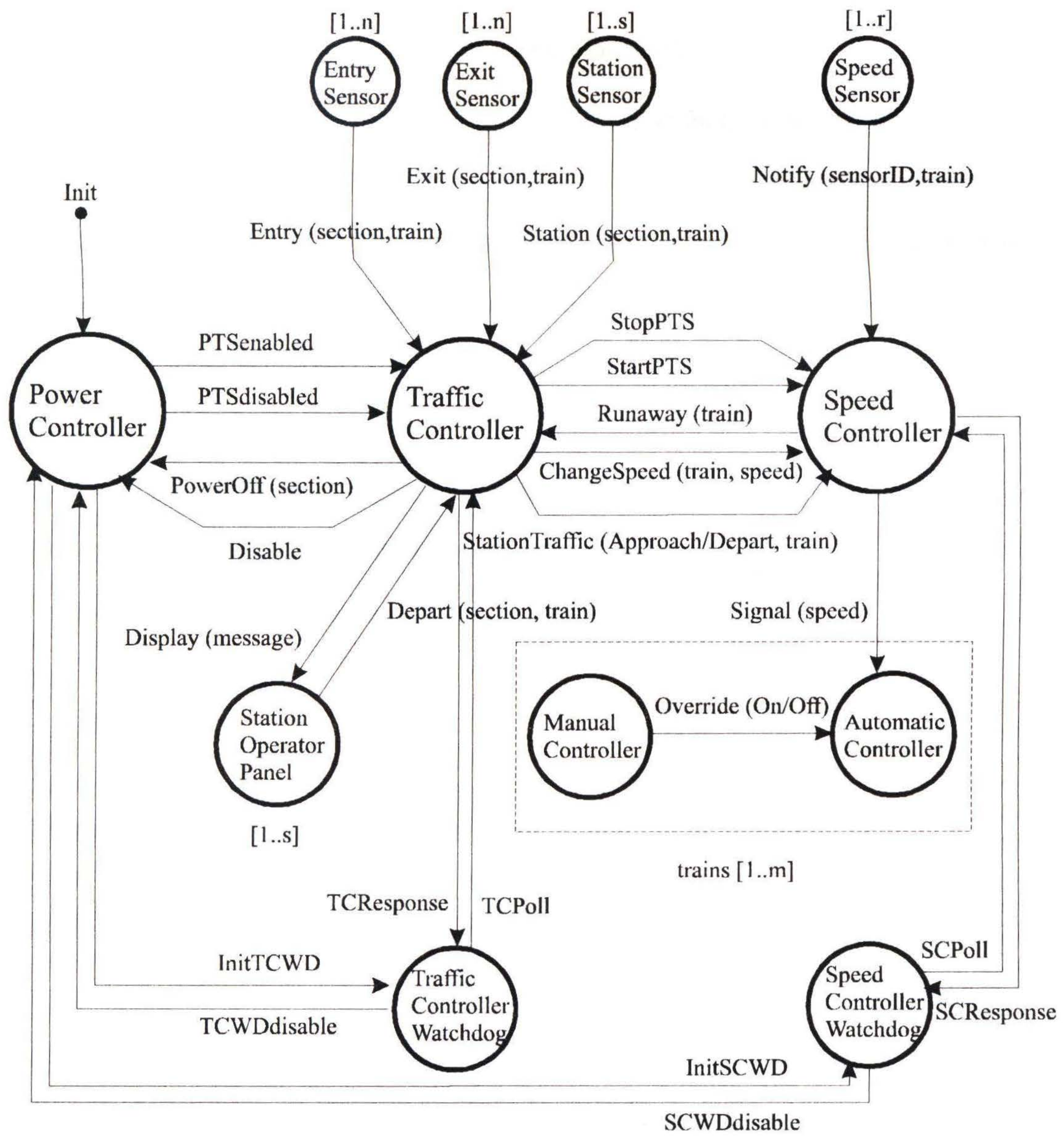


Figure 13. Revised Fault Tolerant High Level CRSMs

7.2 Fault Tolerant Design Modifications

7.2.1 Revised High Level Communicating State Machines

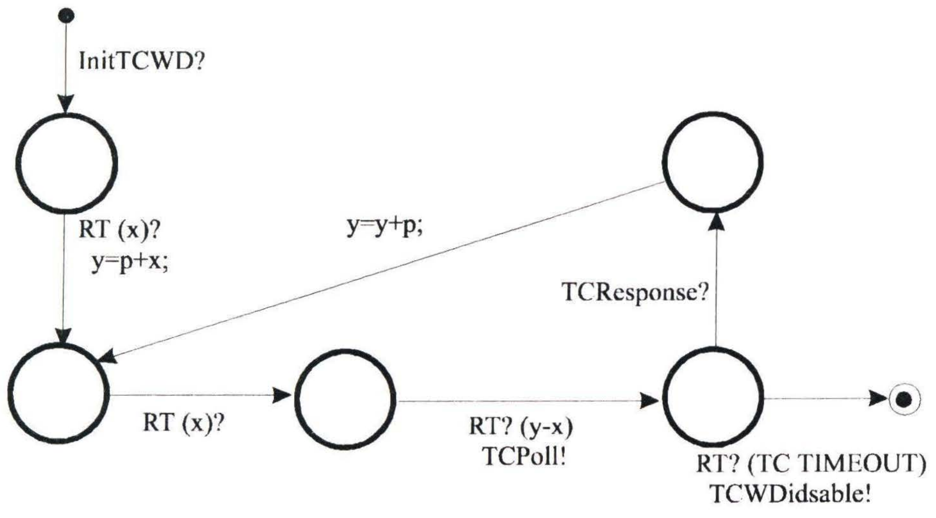
Since the independent, parallel operation of the Watchdog Controller functions are critical to the safety requirements of the fault-tolerant PTS design, a revision of the high level CRSM model is warranted. This revised model should be viewed as a design refinement resulting from the iterative application of failure scenarios. Figure 13 illustrates the revised high level CRSM for the PTS with Watchdog Controllers for the Traffic and Speed Controllers.

Separate Watchdog Controller functions have been added that poll the Traffic and Speed Controllers periodically to determine if they are functioning or if they have failed completely. Upon detection of a failure (a timeout without response to the poll), the PTS is disabled.

7.2.2 State Machine for Watchdog Controller

The Watchdog Controllers monitor the Traffic Control and Speed Controller processes to determine if they are responding. This is accomplished with simple periodic polling of each process. The details of this polling operation are illustrated in the level-2 state machines. The operation of the watchdogs are the same except for the target of the polling requests. The current time is obtained and the periodic interval time, p, is added to it and saved. (The time to perform this operation is deducted from the total timeout period). When the timeout occurs, the target process (either Speed Controller or Traffic Controller) is polled. If a response is received within the timeout period, the polling process continues at the next time interval. If the timeout occurs before the response, then the process is assumed to have failed and a disable message is given to the Power Controller. Figure 14 illustrates the Watchdog Controller state machines.

Traffic Controller Watchdog



Speed Controller Watchdog

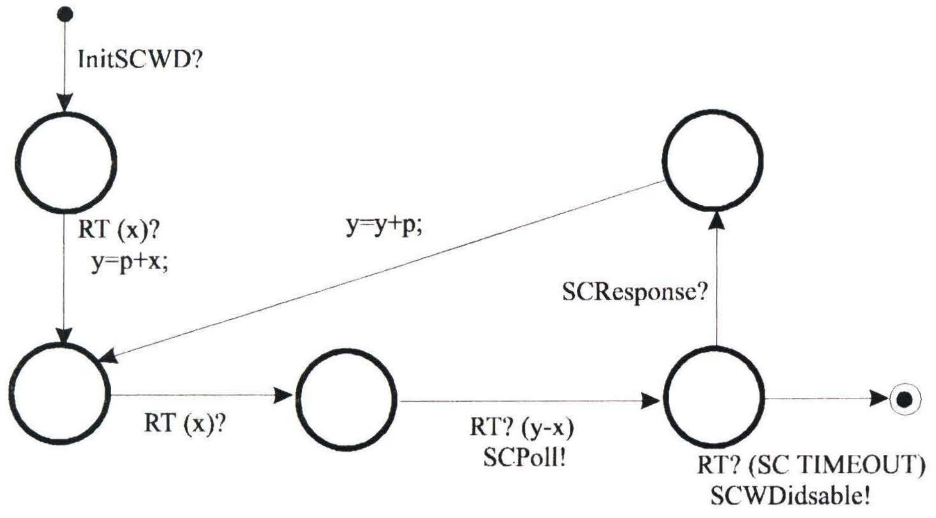


Figure 14. State Machines for Watchdog Controllers

8. PTS Implementation Considerations

Once the fault tolerant failure scenarios have been identified and solutions proposed, the underlying process (task) organization, concurrency, and mapping to hardware can be completed. The specification of object creation and destruction actions can also be included. These implementation-specific characteristics should generally not be added back into the higher level models. They are included as refinements to the models with visually identifiable transitions mapping each higher level refinement to the next lower level. This approach provides a seamless and progressive flow from modeling and design to implementation that is key to ensuring that errors are not introduced from one modeling step to the next.

An abstraction hierarchy [Fern 95] illustrates the organization of concurrent processes and mapping to hardware. The first level of the abstraction hierarchy is the application layer. The application layer illustrates the major concurrent application components of the PTS (the controllers). The physical clustering of applications is included (central power station, PTS control center, and individual trains) to clarify the physical location and consolidation of applications within a single facility. High level communication between the applications is illustrated with dotted arrows. The direction of the arrows indicates the direction of communication (no arrow is 2-way communication).

The Speed Controller and Automatic Controller applications are illustrated with imbedded transmit and receive applications. These are shown at the application layer since they provide the mechanism for remote communication between the Speed Controller application in the Control Center and the Automatic Controllers running in the active trains. The mapping of the transmit and receive applications to processes and hardware is also significant to the fault tolerant characteristics of the PTS design as described in section 7.

The second layer in the abstraction is the fault tolerant layer. The Watchdog Controllers are included in this layer. They are not part of the application layer since they are introduced for fault tolerant design. They are not part of the PTS functional services provided to the user by the application.

The third level in the hierarchy is the process layer. The process layer illustrates the tasks (or process execution instances) that make up the applications. The curved shapes represent tasks (processes) and the attached squares represent the static code sections that are associated with each task. Horizontally oriented arrows indicate the communication between tasks. Each application in the application layer is mapped to the appropriate task or tasks in the process layer. Where one version of the software is used, there is single task. Where multiple versions of the software are used (as illustrated for the Traffic Controller and Speed Controller applications), two separate tasks are shown (a primary and a backup), each with their own unique code section since they are executing different source code. The way the versions are defined in the fault tolerant solutions, the backup task monitors the inputs and outputs of the primary task, so a one-way communication arrow is drawn from the primary to the backup to illustrate this communication. The transmitter functions are triplicated so three separate tasks are shown (X1, X2, and X3). The code sections could be the same for each one. They are illustrated as separate code sections in this abstraction.

The receiver function of the Automatic Controller application includes processes for the mirrored receivers (R1 and R2), the voters (V1 and V2), and the comparator (C1). The communication between the receivers, voters, and comparators are not shown in this diagram due to the additional complexity and confusion that would be introduced by these additional lines.

The machine layer provides an illustration of an option for organizing the processes on specific machines. The machines run independently but machines within the same facility would be connected with common buses or LANs for local communication. External high speed communication links are required between the Central Power Station and the PTS Control Center.

Figure 15 illustrates one example implementation of a mapping from software applications and processes to machines. In this example, the primary versions of the Traffic and Speed Controllers are assigned to machine "Primary" with their code sections in "Primary's" local memory. Similarly, the backup versions are assigned to "Backup" and its local memory. Both watchdog tasks are assigned to the same processor, Watchdog", which is assumed to be capable of multi-processing.

The abstraction hierarchy diagram provides a single view of key application to machine configuration. However, even a simple design problem quickly generates a complex and unreadable diagram. For ease of understanding, the abstraction diagram should focus on the key physical relationships. The details of the underlying implementation for all processes and machines should be included with the detailed process (object) design and implementation documentation.

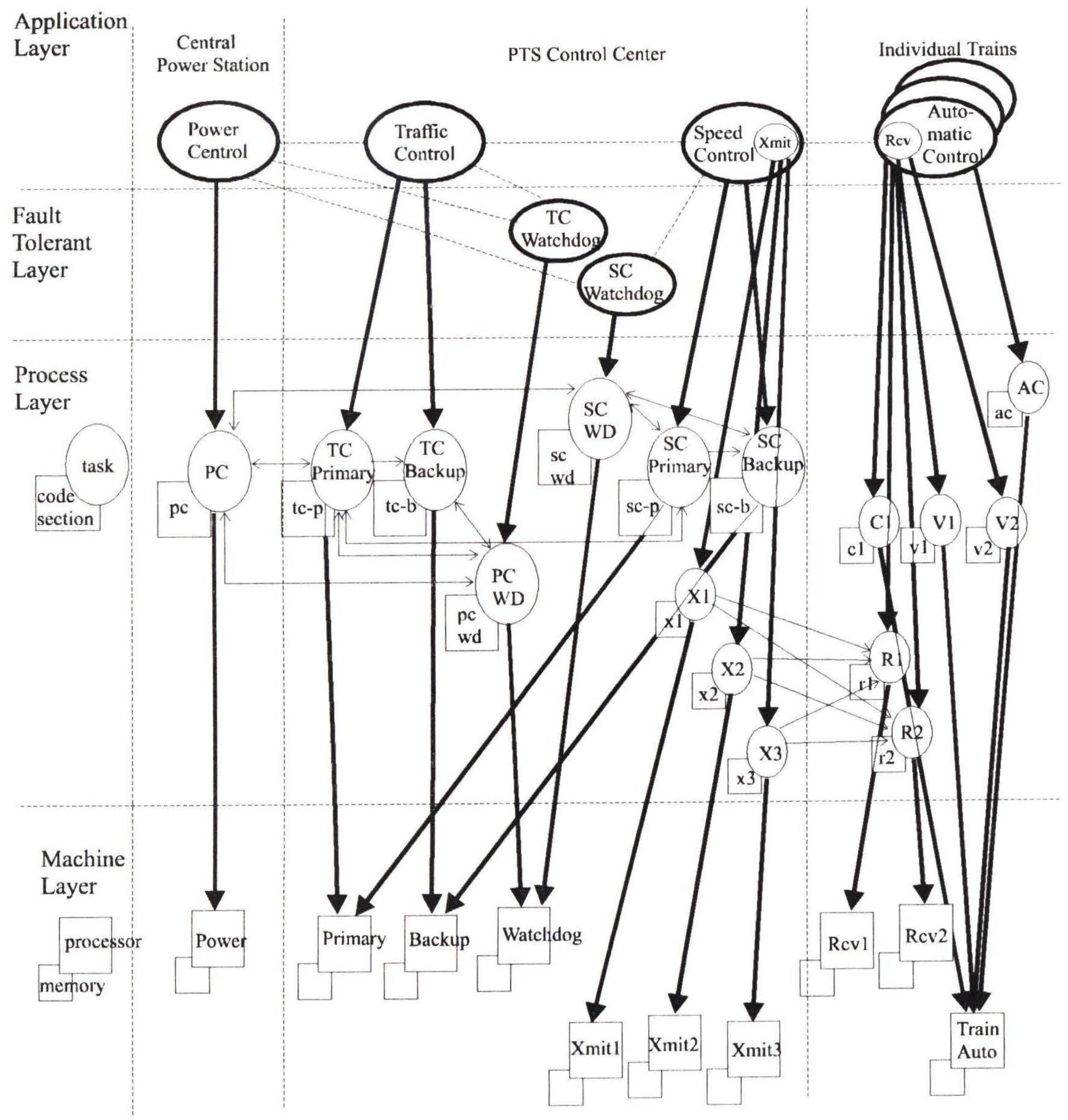


Figure 15. PTS Abstraction Hierarchy

9. Summary and Evaluation

The proposed object-oriented modeling and design methodology, illustrated with the PTS example, is a complete life cycle methodology for developing real-time, fault-tolerant systems. The output of each process step in the methodology is a refinement of a previous step. Since each refinement is incremental, and uses the notations and terminology of the previous step, there is a traceable mapping between steps that can be used for design validation.

The high level models (OMT class diagram and level-1 CRSM) graphically convey the real-world problem and physical machine concurrency that is key to determining whether the high level design is correct with respect to the original problem requirements. The OOA class diagram is derived using conventional OMT methods where the important aspects of the problem are conveyed without the introduction of design and implementation-specific constructs. The OOA class diagram is derived through a series of tries and refinements to obtain the optimal organization of classes and relationships to represent the problem and object structure. Object-oriented classes map conceptually well not only to the real world physical objects but to the concurrency of machines since objects operate independently. Objects communicate with external messaging between other objects. Internal communication is encapsulated as internal operations (or methods). The OOA model is therefore easily mapped to the next step in the methodology: the high level communicating machines.

The key machines in the high level CRSM are easy to identify from the OOA class diagram. These machines are the main controllers for the physical objects represented by the classes. For the PTS, the Power, Traffic, and Speed Controllers were easily identified as concurrent machines requiring external communication channels. Other identifiable machines from the class diagram were the operator panels, the Automatic Controllers on each train, and the sensors.

Proposed Methodology

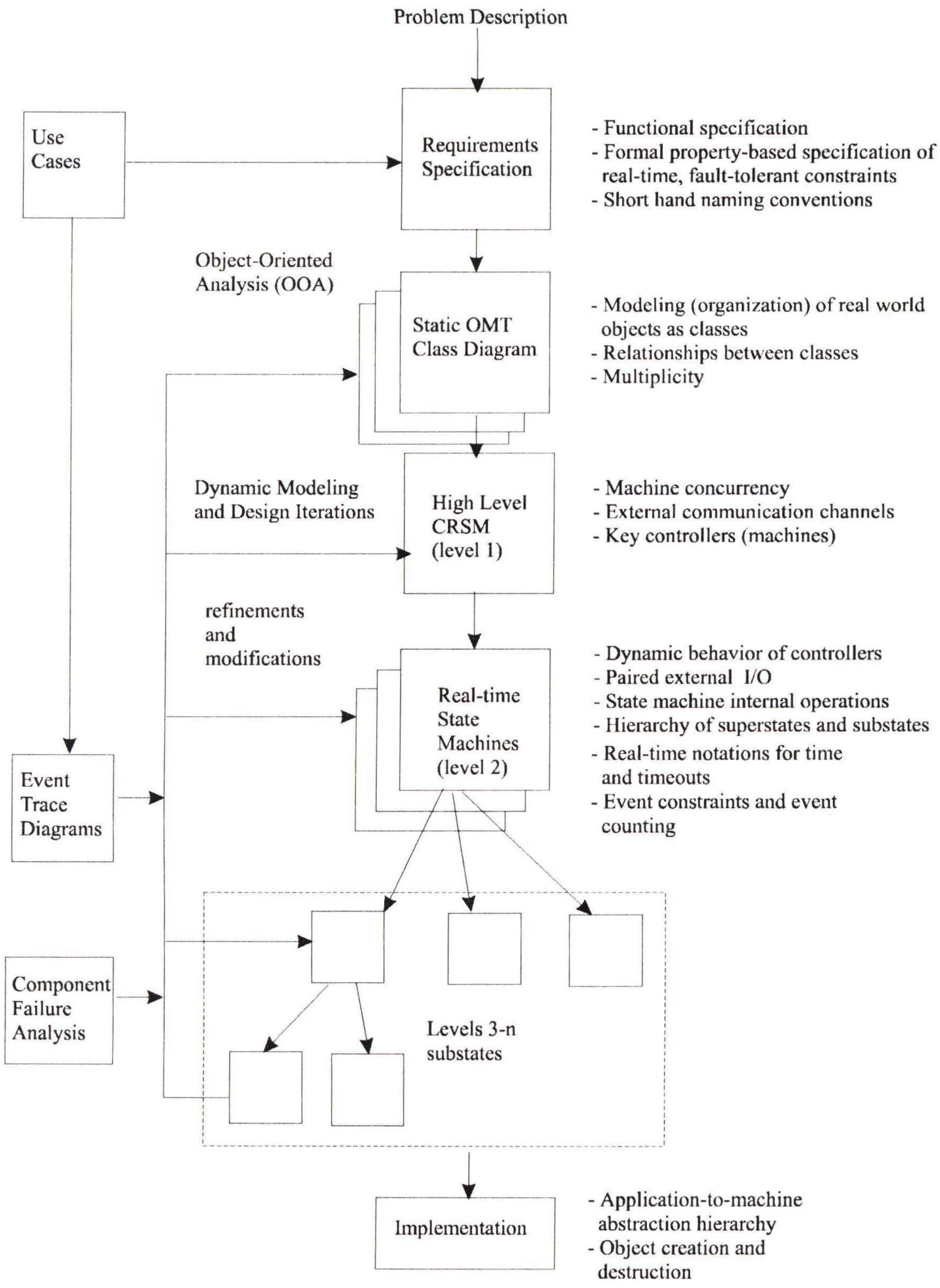

Problem Description
Use Cases
Requirements Specification
- Functional specification
- Formal property-based specification of real-time, fault-tolerant constraints
- Short hand naming conventions
Object-Oriented Analysis (OOA)
Static OMT Class Diagram
- Modeling (organization) of real world objects as classes
- Relationships between classes
- Multiplicity
Dynamic Modeling and Design Iterations
High Level CRSM (level 1)
- Machine concurrency
- External communication channels
- Key controllers (machines)
refinements and modifications
Real-time State Machines (level 2)
- Dynamic behavior of controllers
- Paired external I/O
- State machine internal operations
- Hierarchy of superstates and substates
- Real-time notations for time and timeouts
- Event constraints and event counting
Event Trace Diagrams
Component Failure Analysis
Levels 3-n substates
Implementation
- Application-to-machine abstraction hierarchy
- Object creation and destruction

Figure 16. Proposed Modeling and Design Process Steps

Although machine concurrency and communication channels are conveyed well by the proposed high level models, the real-time aspects of the problem are not conveyed until the lower level models and designs. More precise and formal property-based assertions were used to augment the high level models with these real-time constraints. These formal specifications help map the real-time problem aspects to the low-level design. They also provide a mechanism for validating the resulting implementation. However, in the proposed methodology, there is no one high level graphical model that conveys this complete information. Using the OOA class diagram and the high level CRSM alone, the real-time properties are not easily mapped from the high level model refinements to the lower level designs.

The use of the high level CRSM model as a refinement to the OOA class diagram provides a good transition step to the lower level models and designs. It also visually illustrates the logical concurrent machine configurations requiring communication and synchronization. The high-level machine diagram is easy to understand by software engineers generally familiar with state machine notations. Only simple explanations are required to convey the meaning of the channels and machine concurrency. No dynamic behavior is conveyed with the high level CRSM, only the structure of concurrency and communication.

One of the key benefits of the revised methodology is the combined effect of the CRSM high level structure with the modified notations of the lower level state machines. The multi-level hierarchy of CRSMs provides a mechanism for conveying precisely the sender and receiver pairs for messaging. As with unmodified OMT, other methodologies provide mechanisms for events, but they do not explicitly identify the event as an external directed communication other than by the naming of the event (^target.sendEvent(arguments). Transition events in the revised methodology are easily identified as being associated with receipt of an external message. With traditional OMT state machines, external events on transitions are not easily distinguishable from internal events (operations).

Real time notations are easily conveyed in the revised methodology using the "RT(x) ? [y]" real time expressions and implicit event counter notations. Although these notations require some explanation to be easily understood, they do precisely define the timing and time out behavior of the state machines. Other less formal notations for real-time properties often attempt to convey deadlines and timing (e.g., [event + time], [> time]), but not the underlying state and transition behavior required to map the model to the lower level design. Errors are often introduced by timing-related design issues, especially where race conditions can occur between parallel tasks. Clear designs for timing are critical to the correct operation of the system since timing errors can often go undetected and are difficult to diagnose once introduced into the implementation. The proposed notations for modeling real-time constraints and event handling add more formalism to the state machine dynamic models. This formalism seems to justify the loss in readability in the lower level models.

The evolution of the high level model to low level design is achieved by successive refinements based on normal use case scenarios and their corresponding event trace diagrams. Real time constraints for safety and timing are considered in these scenarios. The more formal property-based real-time constraint specifications and the use cases developed from the problem description can be used to ensure a good mapping from the real-time problem to the lower level design.

Different from CRSMs, the revised methodology permits the use of superstates and substates in the lower level state machines. This improves the readability of the models and designs. It also organizes the functions (and underlying implementation) into separately contained sub-components that could be developed independently. Prototypes of high level superstates can be developed with functions stubbed for the underlying substates.

In the proposed methodology, the initial focus of the analysis and design includes the main operational functions and real-time characteristics specified in the problem description (requirements) and formal specification properties. Normal error handling and safety design issues were considered and included throughout the iteration of the design. The resulting base design met the functional requirements but did not attempt to satisfy the fault tolerant requirements. No attempt was made to design in the fault tolerant characteristics (error detection, failure masking, concurrency/redundancy, repair time) as a part of every design iteration. Instead, component failure analysis (scenarios) were applied following the completion of the base design. Since the details of the base design were known, it was simple to apply the failure scenarios and identify the further design and implementation requirements for fault tolerance. This methodology worked very smoothly and seemed to produce good design results. The results seem consistent with the notion that modularity and concurrency issues should primarily be addressed at the implementation phase of development. The possible negative aspects of this methodology of designing the fault tolerance last are as follows:

- Significant rework and redesign may be necessary. In the PTS example, the decision to go with multiple software versions for the key controllers would significantly impact the controller design. The design of each version would need to include synchronization points, extra communication, and mechanisms for handling the switch over and recovery process upon detection of a failure. Redesign and rework are costly on large projects.

- Much of the hardware cost associated with the application could not have been determined until the end of the design phase since the hardware requirements (machines, redundant sensors, transmitters, and receivers) was not determined until the final design/implementation phase. Cost estimates are typically required early in the project to determine the overall affordability.

- The decision to use multiple versions of the Traffic Controller and Speed Controller software at the end of the design cycle would be very costly and would likely delay the implementation of the entire project. Software development often incurs the highest costs and has the longest lead time requirements for development. As a result, the decision to build multiple software versions should be made as early as possible in the design process.

As a part of the transition from design to implementation, an abstraction hierarchy was provided to map the applications to configurations of tasks and actual machine configuration. These issues must certainly be determined at the implementation level. The abstraction hierarchy diagram, although conceptually sound, quickly resulted in a complex diagram that would likely be unreadable for a large, complex project.

An important step in mapping the design to implementation is the consideration of the life of objects in the system. Object creation and destruction design and implementation considerations are often overlooked until the final coding begins. Errors can be introduced into the system by poor design of the actions associated with object creation and destruction. Harel's O-Charts introduce the design of these operations as part of the modeling and high level design. Although this creates a better mapping from design to implementation, it does tend to introduce implementation-specific constructs quite early in the model, adding undesirable complexity and taking focus away from the key modeling and design issues . Object creation and destruction should be included in the methodology as an important low level design step, but should not be reflected in the high level models.

Table 5 provides a comparison of the proposed methodology with selected representative methodologies. For each methodology, the high level model and dynamic model approaches are compared. The key real-time, fault-tolerant aspects of each methodology are also provided.

	Proposed Methodology (Thesis)	**Requirements State Machine (RSM)** [Leve 94]	**Statecharts** [Hare 90] [Hare 96B]	**O-Charts** [Hare 96A]
High Level Analysis Model	OMT class structure diagram, High level CRSM for machines and communication	High level system component diagram for machines and communication	Activity chart (activities and data flow)	O-Charts that model the structure of classes (OMT-like)
Dynamic Model	Modified low-level CRSMs for behavior of controllers	Modified Statecharts with interface specifications	Statecharts with AND composition of state machines, broadcast events	Statecharts for behavior of classes, object creation and destruction
Communication between machines	Explicit pairing of external send and receive events between machines	Paired SEND and RECEIVE external events between physical components	No distinction between internal and external events	Specification of externally queued events vs. operations
Timing constraints	Explicit notations for expressing real time and timeouts in dynamic model	Timing expressed with event constraints in dynamic model	Timeout events and scheduled actions expressed in step execution rules	Timeout events and scheduled actions expressed in step execution rules
Fault-tolerant considerations	Component failure analysis, Design changes to models	Model includes component failures, No design method	Not addressed	Not addressed
Implementation	No tools or mapping to C++	RSML simulator	Statemate to generate Ada or C	O-Mate to generate C++

Table 5. Comparison of Methodologies

10. Conclusions

This thesis proposes a complete life cycle methodology for the modeling and design of real-time, fault-tolerant systems. Although based upon the well-known OMT concepts and process steps, unique process steps enhance the traditional OMT for real-time, fault-tolerant systems.

- Property-based formal specifications were included to ensure that the system specification accurately describes the critical constraints and behavior. These formal specifications make the specification more precise and can be utilized to verify the timing and functions further defined in the dynamic model and design.

- High level communicating real-time machine modeling was added as a refinement of the static class diagram. This is a new step in the methodology that models the important physical machine concurrency and external communication paths. This CRSM model maps easily from the OMT OOA model. It also provides an easy transition to the modeling and design of the communicating state machines.

- New notations were introduced to convey the real-time behavior and events of the dynamic model (communicating state machines). These notations borrow from CRSMs and other real-time notation extensions and are modified for consistency with other OMT dynamic modeling notations. External events are modeled with directed, paired events. Event counting notations and short hand naming conventions for constraints make the state machines easy to ready and understand while accurately conveying the real-time design. Real time and timeouts are accurately and precisely conveyed using the CRSM real time notations. Unlike CRSMs, the proposed methodology preserves the ability to nest substates to reduce the complexity of the higher level state machines.

- A component failure analysis method is proposed for introducing fault tolerance into the design. By performing the failure analysis after the initial modeling and design is completed, the fault tolerant aspects are kept separate from the key operational functions of the system. Methodologies that introduce fault tolerance earlier in the modeling process blend the functional and fault tolerant components so that is difficult to understand the rationale for the design elements.

- This thesis also provides a unique, hypothetical example of a possible complex real-time system. The example is used to illustrate the process steps of the methodology and was used to refine the methodology as the example was created.

10.1 Future Considerations

Harel's statecharts are based upon state machine notations and provide some advantages over traditional state machines in their ability to show concurrent state machines as AND-states. This approach could have simplified some aspects of the PTS state machines. For example, the Traffic Controller and Speed Controller could be illustrated in one statechart as AND-states. An enable message from the Power Controller could have been provided as a single transition, initiating both states. A methodology combining OMT OOA class diagrams, high level CRSMs, and statecharts with modified real-time notations might provide a simpler modeling of the system. At the transition to implementation, the statecharts could be refined to include implementation-specific O-chart notations that would map better to a C++ implementation than the proposed methodology.

References

[Awad 96] M. Awad, J.Kuusela, J. Ziegler, *Object-Oriented Technology for Real-Time Systems*, Prentice-Hall, 1996.

[Burn 94] A. Burns, A. Wellings, "HRT-HOOD: A Structured Design Method for Hard Real-Time Systems", *Real-Time Systems,*", volume 6, number 1, January 1994, pp. 73-114.

[Cole 92] D. Coleman, F. Hayes, S. Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design", *IEEE Transactions on Software Engineering*, volume 18, number 1, January 1986, pp. 9-18.

[Chon 95] M. J. Chonoles, C. C. Gilliam, "Real-time Object-Oriented System Design Using the Object Modeling Technique (OMT)*"*, *Journal of Object Oriented Programming*, June 1995, pp. 16-24.

[Farn 96] F. Jahanian, A. Mok, "Modechart: A Specification Language for Real-Time Systems", *IEEE Transactions on Software Engineering*, volume 20, number 12, December 1994, pp. 933-947.

[Fern 95] E. Fernandez, R. France, "Formal specification of real-time dependable systems", *Proceedings of the International Conference on the Engineering of Complex Computer Systems"*, IEEE 1995.

[Goma 86] H. Gomaa. "Software Development of Real-Time Systems", *Communications of the ACM,* volume 29, number 7, July 1986, pp. 657-668.

[Hall 96] A. Hall, "Using Formal Methods to Develop an ATC Information System", *IEEE Software*, March 1996, pp. 66-76.

[Hare 90] D. Harel, H. Lachover, A. Naamad, A. Phueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, volume 16, number 4, April 1990, pp. 609-620.

[Hare 87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, volume 8, 1987, pp. 231-274.

[Hare 96A] D. Harel, E. Gery, "Executable Object Modeling with Statecharts", *Proceedings of the 18th International Conference on Software Engineering*, IEEE Press, March 1996, pp. 246-257.

[Hare 96B] D. Harel, A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Transactions on Software Engineering and Methodology*, volume 5, number 4, October 1996, pp. 293-333.

[Heit 95] C. Heitmeyer, D. Mandrioli, "*Formal Methods for Real-Time Computing: An Overview*", John Wiley & Sons, 1996.

[Kavi 92] K. Kavi, S. Yang, "Real-Time Systems Design Methodologies: An Introduction and a Survey", *The Journal of Systems and Software*, April 1992, pp. 85-89.

[Kras 95] G. Krasovec, M.Baker, S. Gheorghe, "Target-Tracking: A Real-Time Object Oriented

Design Experiment", *Proceedings of the International Conference on the Engineering of Complex Computer Systems",* IEEE 1995, pp. 290-297.

[Merc 90] C. Mercer, H. Tokuda, "The ARTS Real-Time Object Model," *Proceedings of 11th IEEE Real-Time System Symposium,* 1990, pp. 2-10.

[Leve 94] N.G. Leveson, M.P.E. Heimdahl, "Requirements Specification for Process-Control Systems*",* *IEEE Transactions on Software Engineering,* volume 20, number 9, September 1994, pp. 684-707.

[Rati 96] Rational Inc., "Unified Modeling Language for Real-Time Systems Design", www.rational.com, 1996.

[Rumb 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object Oriented Modeling and Design*, Prentice Hall, 1991.

[Rumb 93] J. Rumbaugh, "Controlling Code - How to Implement Dynamic Models", *Journal of Object Oriented Programming*, May 1993, pp. 25-30.

[Rumb 94A] J. Rumbaugh, "The Life of an Object Model", *Journal of Object Oriented Programming*, March-April 1994, pp. 24-32.

[Rumb 94B] J. Rumbaugh, "Building Boxes: Subsystems", *Journal of Object Oriented Programming*, October 1994, pp. 16-21.

[Rumb 95] J. Rumbaugh, "OMT: The Dynamic Model", *Journal of Object Oriented Programming*, February 1995, pp. 6-12.

[Seli 92] B. Selic, G. Gullekson, J. McGee, I. Engelberg, "ROOM: An Object-Oriented Methodology for Developing Real-Time Systems", *Proceedings on the Fifth International Workshop on Computer-Aided Software Engineering*, 1992, pp. 230-240.

[Shaw 92] A. C. Shaw, "Communicating Real-Time State Machines", *IEEE Transactions on Software Engineering,* volume 18, number 9, September 1992, pp. 805-816.

[Shaw 95] M. Shaw, "Comparing Architectural Design Styles", *IEEE Software,* November 1995, pp. 27-41.

[Ward 86] P. Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing", *IEEE Transactions on Software Engineering,* volume 12, number 2, February 1986, pp. 198-210.