UC Berkeley
Research Reports

Title
Safety Analysis Of Automated Highway Systems

Permalink
https://escholarship.org/uc/item/8xn29461

Author
Leveson, Nancy G.

Publication Date
1997

eScholarship.org Powered by the California Digital Library
University of California

This paper has been mechanically scanned. Some errors may have been inadvertently introduced.

CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Safety Analysis of Automated Highway Systems

Nancy G. Leveson

California PATH Research Report
UCB-ITS-PRR-97-36

This work was performed as **part** of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

The contents of this report reflect the views of the authors who are responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California. This report does not constitute a standard, specification, or regulation.

October 1997

ISSN 1055-1425

Final Report2 : Safety Analysis of Automated Highway Systems

Nancy G. Leveson
University of Washington

1 Executive Summary

This is a second final report on the work done under this grant. A first report was written and submitted that included a description and copies of publications under grant funding. At the time, however, I did not know that a separate report was required because the grant number had changed during the 3-year period of the grant. This second report contains a more detailed description of the safety analysis techniques and tools that have been developed for a state-based requirements specification language called Requirements State Machine Language (RSML). These tools include a simulator that allows for forward and backward execution of RSML specifications, a fault tree generator that is based on backward simulation, and tools to check for consistency and completeness of specifications. An example requirements specification of an automated highway system design is described and the functionality of the tools are demonstrated on the model. The report also contains a copy of a dissertation on a new safety analysis technique, called Software Deviation Analysis (SDA) that was partially supported by this grant. The technique allows analysis of the behavior of the model in the presence of deviations from expected inputs, i.e, how the system would work in an imperfect environment.

Safety Analysis Tools for AHS Models

1 Introduction

In many systems, especially safety-critical systems, it is important to specify the required behavior of the system as completely and unambiguously as possible. Incorrectly behaving software can have disastrous consequences. It is at this stage of system specification that some of the most costly errors are introduced because accidents are primarily related to specification, rather than implementation, errors [13, 8]. These errors are the last and most difficult to find according to most studies of software errors. It is thus desirable to provide readable, unambiguous, and complete requirements specifications and to be able to perform an analysis of the specifications to validate desired properties of the system.

The goal of our work is to explore the limits of automated analysis to provide information useful in safety-critical project development. We are exploring various types of analyses that can be performed on state-machine models. Although these ideas can be adapted to most state-machine modeling languages, the language used in this paper is Requirements State Machine Language (RSML), which was developed to specify the system requirements for TCAS II (Traffic Alert and Collision Avoidance System) for the FAA [11]. This language includes many of the hierarchical abstraction and parallel state-machine features of modern state-machine specification languages [3]. These features make such languages feasible for specifying complex systems, but they sometimes also greatly complicate the analysis process. We assume that the reader is familiar with the basic features of such languages, but we include a section describing the features of RSML that are relevant to this paper.

For this grant, we have developed an RSML specification of an automated highway system (AHS) to demonstrate the usefulness of this approach to evaluating safety of AHS systems. It is important to note that our specification is used to demonstrate the analysis tools and approach, and the model itself does not purport to be complete or based on any real AHS design. We, however, believe it to be non-trivial and realistic enough to provide a convincing demonstration.

The rest of this paper is organized as follows. Section 2 describes the AHS model. Section **3** presents the basic features of RSML relevant to this paper. Finally, Section **4** describes the safety analysis techniques, including forward and backward simulation, generation of fault trees, and consistency and completeness analysis. The complete model is contained in the appendix.

2 The AHS Example Model

In this paper, we have used RSML to create a requirements specification of an AHS model. The specification is based some of the fundamental design decisions in the AHS model described in [5, 6]. We have chosen to concentrate on modeling the motion of vehicles on the automated highways.

Our AHS model consists of a highway with multiple automated lanes in which traffic is organized in platoons of closely spaced vehicles under automatic control. The "intelligence"in the model is concentrated in the vehicle control systems and in the roadway infrastructure. The characteristics of the model are as follows :

- The highway consists of multiple lanes, each lane supporting vehicles traveling in the same direction.
- The entry and exit of vehicles to or from the highway or vehicle entry checks are not included in the model.
- Vehicles move on the highway by performing three kinds of maneuvers: *Change-lane, Merge,* and *Split* (described below).
- On the highway, all vehicles move at the same speed except when they take part in one of the above-mentioned three maneuvers.
- Roadside information structures are present along the highway. They sense traffic conditions and communicate this information to the vehicles.
- Vehicles travel in platoons, i.e., groups of closely spaced vehicles. Each platoon has a platoon leader, which is defined as the vehicle in the front of the platoon. The number of vehicles in a platoon can vary from one to a specified maximum. Each platoon must maintain a headway from the others platoons. Furthermore, each vehicle in a platoon (except for the platoon leader) must be separated from the one in front of it by a constant distance. These limits and assumptions are designed to enable the system to optimize capacity and reduce travel times of vehicles on the highway.

- Vehicles are provided information on the speed and position of other vehicles around them through sensors and through communication with the roadside information structures and the other vehicles.

The model includes three classes of maneuvers that a vehicle can perform on the automated highway: (i) *Change-lane,* (ii) *Merge,* and (iii) *Split. Change-lane* enables a single vehicle to move into an adjacent lane, *Merge* enables a platoon to join with the platoon in front of it to form a single platoon, and *Split* enables a platoon to separate into two. *Change-lane* is performed by a vehicle that is the only vehicle in its platoon, after ensuring, through communication, that no vehicles are present in the adjacent lane that could impede its maneuver. *Merge* is performed by a platoon by accelerating toward the platoon directly ahead of it until it becomes part of that platoon. In *Split,* either a leader of a platoon can split from the platoon by accelerating away from it (after ensuring that it is safe to do so), or part of the platoon (all vehicles in the platoon in front of the vehicle that initiated the maneuver) can accelerate away to form a separate platoon.

3 RSML specification of the AHS Model

RSML is based on an underlying Mealy machine and adopts some of the features introduced in Statecharts [3], including hierarchical abstraction into superstates and parallel state machines. A specification may be composed of multiple components, where each component specifies the behavior of a corresponding system component. A more detailed description of RSML can be found in [11].

The AHS can be modeled using multiple identical sub-systems, each sub-system representing a vehicle or a roadside control structure. The environment for each vehicle consists of the other vehicles on the highway as well as roadside controllers along the highway. Each vehicle can be considered as consisting of various components: the sensors (which provide information about other vehicles in the vicinity), the controller (which is responsible for the maneuvers of the vehicle on the highway), a transmitter (which can send messages to other vehicles), a receiver (which can receive messages from other vehicles), and others.

The vehicle controller handles communication with the vehicle's environment, and it controls the maneuvers in which the vehicle can take part. We have specified the behavior of this component using RSML. The state machine model of the controller is an abstraction of the perceived behavior of the controller and can be iteratively modified during the requirements development

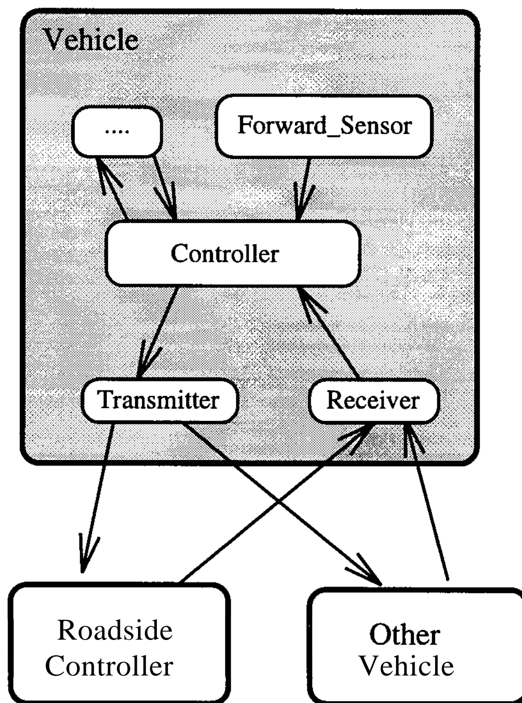


Figure 1: *Communicating components in the AHS model. Arrows represent communication between components.*

		OR	
A	Position **In State** Single	·	T
N	Distance **In State** IP	T	T
D	ThisLaneFront	T	·
	OwnSpeed = System_Speed	F	T

Figure 2: *An **AND/OR** table.*

phase as the understanding of the environment and the controller behavior changes. Specifications can be made more detailed or abstract, depending on the kind of analysis desired.

RSML components can communicate with each other, or with their environment, through point-to-point messages over defined channels. RSML messages are received asynchronously and queued upon arrival. The interfaces are connected to specific communication channels where the receipt of a message on a channel can set variable values and trigger events. Each channel is connected to one input interface and one output interface, and each interface is connected to exactly one channel.

Within a vehicle in the AHS, the sensor and the receiver components provide inputs to the controller component, while the controller provides inputs to the transmitter, which in turn communicates with the receiver on other vehicles. This communication structure is shown in Figure 1. Within a component, internal events are broadcast and available everywhere.

RSML components contain a state hierarchy, transitions between states, a set of input and output interfaces, a set of variables and constants, and a set of events to order the transitions.

Each transition in RSML has a source, destination, trigger event, and events that it triggers along with a guarding condition that must be true for the transition to be taken. RSML provides the full predicate calculus for expressing guarding conditions: A guarding condition may be either a simple Boolean TRUE or **FALSE**, an AND/OR table, or an existential or universal quantifier of a variable over another condition.

An AND/OR table is a disjunction consisting of Boolean expressions, which may contain macros (other AND/OR tables, that is, functions returning a Boolean value) or predicates over arithmetic expressions (including numeric functions and variable and table references). An example of an AND/OR table is shown in Figure 2. The far left column of the AND/OR table lists the logical phrases; each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its

elements are true. A dot denotes "don't care".

We have found while observing the use of RSML by application experts that AND/OR tables provide a more natural and reviewable notation when compared to other formal notations, such as the usual predicate calculus notation. A predefined macro, IN-STATE, returns true if the component is in a specified state. A predefined function, TIME, when applied to a variable or event returns the time that the variable was last assigned or the event was triggered. It is also possible to retrieve the nth last time that a variable was assigned or event was triggered. Thus, the condition represented by the AND/OR table in Figure 2 will evaluate to true if either (1) Distance is in state IP, ThisLaneFront is true, and the variable OwnSpeed does not have the same value as the constant System-Speed, or (2) Position is in state Single, Distance is in state IP, and OwnSpeed has the same value as System-Speed.

We describe the state machine model of the controller in the next subsection. We then describe the rest of the RSML specification of the controller in terms of vehicle maneuvers. We finally provide a detailed example of the Merge maneuver to show some of the transitions and interfaces in the specification.

3.1 State description of the Model

In the RSML model shown in Figure **3,** the controller component is modeled as four parallel state-machines. This state decomposition represents one of many different ways in which the controller can be specified.

The Maneuver-Status state machine represents the maneuver in which the vehicle is presently engaged. It is composed of the following atomic states :

- No_Maneuver: Vehicle is currently not engaged in any maneuver,
- Merge: Vehicle is part of the Merge maneuver,
- Change-Lane: Vehicle has initiated the Change-lane maneuver,
- Split: Vehicle has initiated the *Split* maneuver,
- Busy: Vehicle is participating in a maneuver, but did not initiate that maneuver,
- Waitl: Vehicle is waiting for a reply from another vehicle, and
- Wait2: Vehicle has received a reply from one vehicle and is waiting for a reply from another.

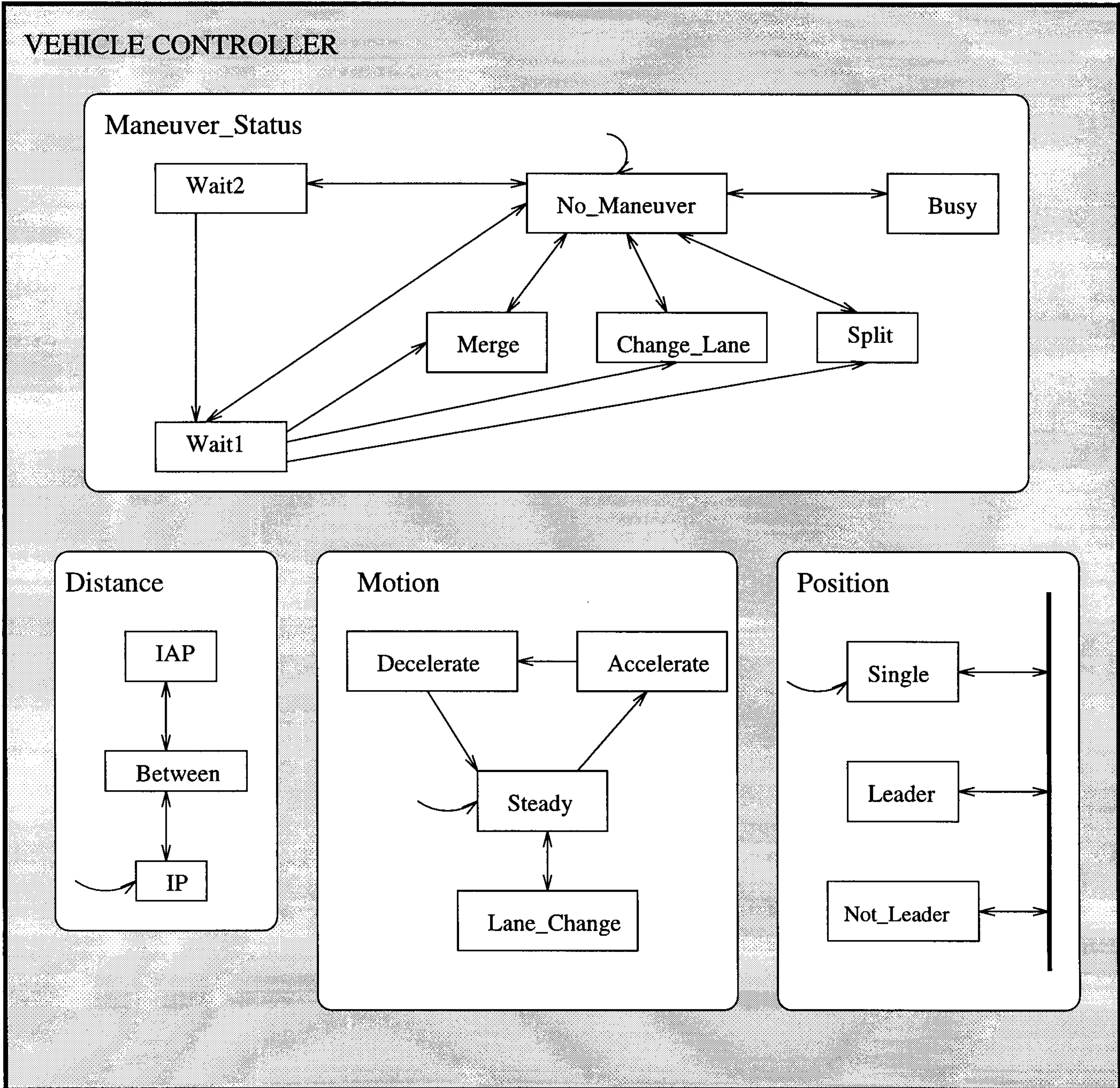


Figure **3:** *The state-machines comprising the controller component. Arrows between states denote the presence of at least one transition between them.*

The Distance state machine represents the distance between the vehicle and the nearest one ahead of it and in its lane:

- The IP (inter-platoon distance) indicates that the distance to the closest vehicle in front is at least the minimum desired between platoons. Such a situation can arise, for example, when the vehicle is the leader of a platoon and is not participating in any maneuver.

- IAP (intra-platoon distance) indicates that the closest vehicle in front is at a distance equal to or less than the desired distance between vehicles in a platoon. Such a situation can arise, for example, when a vehicle is part of a platoon but not the leader of that platoon.

- Between indicates that the closest vehicle in front is neither far (at an inter-platoon distance) nor close (at an intra-platoon distance) but in between. Such a situation can arise, for example, when a vehicle is the leader of a platoon that is merging with the platoon ahead.

The Motion state machine represents the speed at which the vehicle is moving:

- Steady: Vehicle is moving at a constant speed, equal to the speed of vehicles on the highway that are not participating in a maneuver,

- Accelerate: Vehicle is accelerating; for example, at the beginning of a *Merge* or a *Split,*

- Decelerate: Vehicle is decelerating; for example, at the end of a *Merge* or a *Split,* and

- Lane-Change: Vehicle is changing lanes in a diagonal motion.

The Position state machine represents the position of a vehicle within a platoon:

- Single: Vehicle is the only one in the platoon,

- Leader: Vehicle is the leader of the platoon, and there is at least one more vehicle in that platoon, and

- Not-Leader: Vehicle is part of the platoon, but not the leader

3.2 Rest of the specification

The rest of the RSML specification consists of constants, events, input and output variables, input and output interfaces, and descriptions of the transitions between states. The input and output interfaces represent messages between the `Controller` component and its environment (other components or vehicles or roadside control structures). As mentioned earlier, the RSML specification was developed based on three maneuvers that describe the motion of vehicles on the automated highway. These maneuvers are now described in detail. The roadside controllers decide if it is feasible for a vehicle to initiate a maneuver, and a message is sent to the appropriate vehicle accordingly.

3.2.1 The *Change-lane* maneuver

A *Change-lane* can be initiated only if there are no other vehicles in the platoon. Before changing lanes, the vehicle must ensure that there are no other vehicles within a critical distance (such that they would impede the changing of lanes) in the other lane. Assuming that vehicle A wants to change to lane L, three situations arise :

- There is no vehicle in L within a critical distance of A. In this case, A can change lanes without communicating with any other vehicles.
- There is at least one platoon in L within a critical distance of A (either ahead or behind it), while there is no platoon within a critical distance in the other direction. In this case, A must communicate with the platoon in L closest to it before deciding to change lanes. Suppose B is the leader of the closest platoon in L. On receiving a message from A, if B is not taking part in some other maneuver and it is feasible for A to change lanes without colliding with any vehicle in B's platoon, B agrees to take part in A's *Change-lane* and continues to move at a steady speed (along with the rest of the vehicles in its platoon) while A changes lanes.
- There is at least one platoon in L ahead of A and at least one behind A that are within a critical distance. In this case, A needs to communicate with the closest platoons on both sides of it before deciding to change lanes. Suppose B and C are the leaders of these two closest platoons. Then, as in the previous case, if B and C are not taking part in some other maneuver and it is feasible for A to change lanes without colliding with a vehicle in either of their platoons, both agree to A's *Change-Lane,* thus continuing to move at a steady speed (along with the rest of their platoons) while A changes lanes.

After A changes lanes, it continues to be a single vehicle platoon.

3.2.2 The *Merge* maneuver

A *Merge* combines two successive platoons in a single lane into one. *Merge* is always initiated by the leader of the rear platoon, which accelerates towards the platoon in front and merges into it. Merging is subject to the condition that the combined size of the two platoons does not exceed the limit for platoon size.

Assume that vehicle A is the leader of the rear platoon, i.e. the platoon that wants to merge, and vehicle B is the leader of the platoon with which A's platoon wants to merge. If B receives a message from **A** and is not involved in another maneuver, then B ascertains whether a merge is feasible by checking that the sum of the number of vehicles in the two platoons involved does not exceed the limit for a platoon size. If B agrees to the merge, then A and the rest of the vehicles in A's platoon merge with B's platoon by accelerating toward it and then subsequently decelerating.

3.2.3 The *Split* maneuver

A *Split* involves separating a platoon into two. It may be needed, for example, because a platoon size has exceeded the limit or because a vehicle in the platoon needs to change lanes. *Split* may be initiated by any vehicle in the platoon. Two situations arise :

- The leader of a platoon decides to split. A split can take place if there is no vehicle within a certain critical distance in front of the leader (so as to impede its movement). In this case, the leader accelerates away from the rest of the vehicles in its platoon until it reaches a safe distance.

- A vehicle that is not the leader of a platoon decides to cause a split. **A** split can take place if there is no vehicle within a certain critical distance in front of the leader of that platoon. In this case, all vehicles in front of the initiator accelerate away from it to form their own platoon.

3.3 Detailed specification of *Merge*

In order to demonstratre the use of RSML to specify the controller, some of the transitions and communication (interfaces) involved in *Merge* are now described. They represent the RSML specification of the AHS that forms the

input to the analysis tools. Only part of *Merge* is shown here. The entire maneuver is described in Appendix A.

Assume A is the vehicle that attempts to initiate *Merge* while B is the leader of the platoon in front of it (if any). A initiates *Merge* upon receiving an appropriate message from the environment (a roadside control structure). This message is represented as part of an input interface of A.

Interface Sys_M_start:
Source: Roadside_Control_Structure
Trigger Event: RECEIVE(Vehicle1_id, Dist_ahead)
Condition:
Output Action: Systemmerge
Description: Receive indication from roadside control structure to initiate *Merge*. Vehiclelid is id of the leader of the platoon with which we will merge. Dist_ahead is the distance between the two platoons.

Upon receiving an indication to start *Merge,* A checks if it is capable of initiating the maneuver. If so, it initiates the sending of a message to B and waits for an acknowledgement.

Transition(s): No_Maneuver ⟶ Wait1

Location: Controller
Trigger Event: Systemmerge
Condition:

		OR	
AND	Position **In State** Single	·	T
	Position **In State** Leader	T	·
	Distance **In State** IP	T	T
	Motion **In State** Steady	T	T
	Dist_ahead $\geq$ IP_DISTANCE	T	T

Output Action: Reqmerge, Set-vehiclelid, Set_own_id,Set_num_vehicles_p, Set_dist_ahead
Description: Send request to B to merge with it. Output action will result in a message to B to request a merge. Generate events that trigger assignments to relevant output variables.

A sends the message through an output interface, indicating the vehicle id of B, the number of vehicles in its platoon, and how far behind it is. Note that the sending of the message is triggered by `Req_merge`, which was generated as a result of the transition **[NoManeuver ⟶** `Wait1]`.

Interface M_send:
Destination: Vehicle
Trigger Event: Req_merge
Output Action: SEND(Vehicle1_id, Own_id, Num_vehicles_in_platoon, Dist_ahead)
Description: Send message to leader of platoon ahead, indicating desire to merge.

A waits for a reply from B. If A does not receive any reply from B within a specified time period (because, for example, B is busy in another maneuver, or there is a communications failure), A times out and aborts the maneuver.

Transition(*s*): Wait1 ⟶ No_Maneuver

Location: Controller
Trigger Event: TIMEOUT (TIME(Req_merge), TIMEOUT_VALUE)
Condition:

AND / *OR*

Condition	OR	
Position **In State** Single	·	T
Position **In State** Leader	T	·
Distance **In State** IP	T	T
Motion **In State** Steady	T	T

Output Action:
Description: No response received from leader of platoon ahead within time limit. Abort maneuver.

If A does receive a reply from B (indicating B's approval of the maneuver), it starts its *Merge*. (B sends a similar message to other vehicles in **A's** platoon also.)

Interface M_rcv_ok:
Source: Vehicle
Trigger Event: RECEIVE(Vehicle1_id, Num_vehicles)
Condition:
Output Action: Rcvd_merge_ok
Description: Receive message from leader of platoon in front indicating *Merge* is OK. Num_vehicles is the number of vehicles in the platoon ahead.

Transition(s): Wait1 ⟶ Merge

Location: Controller
Trigger Event: Rcvd_merge_ok
Condition:

AND	OR	
Motion **In State** Steady	T	T
Position **In State** Leader	T	·
Position **In State** Single	·	T
Distance **In State** IP	T	T

Output Action:
Description: Leader received indication to proceed with merging with platoon ahead.

All vehicles in A's platoon start accelerating towards B's platoon.

Transition(s): Steady ⟶ Accelerate

Location: Motion
Trigger Event: Rcvd_merge_ok
Condition:

AND	OR		
Position **In State** Leader	T	·	·
Position **In State** Single	·	T	·
Position **In State** Not-Leader	·	·	T
Maneuver-Status **In State** Wait1	T	T	·
Maneuver-Status **In State** No-Maneuver	·	·	T
Distance **In State** IP	T	T	·
Distance **In State** IAP	·	F	T

Output Action:
Description: Start accelerating towards platoon ahead to merge with it.

Transition(s): IP ⟶ Between

Location: Distance

Trigger Event: Rcvd_merge_ok

Condition:

AND	OR	
Position **In State** Single	·	T
Position **In State** Leader	T	·
Maneuver_Status **In State** Wait1	T	T
Motion **In State** Steady	T	T
Dist_ahead = IPDISTANCE	T	T

Output Action:

Description: Start merging with platoon ahead. This is the case where the leader of the merging platoon is IPDISTANCE behind the platoon ahead.

Each vehicle knows the distance to traverse in order to join B's platoon at the rear. Each vehicle accelerates for half this distance and decelerates (at the same rate) for the other half in order to merge with B's platoon at the same relative speed. (The transitions involving the deceleration of the vehicles are not shown.) Once A's platoon has merged with B's, all vehicles in the former revert to their default states. A is no longer the leader of its platoon. A also lets B know that *Merge* is complete. B receives an indication of the completion of the maneuver, and its states are reset accordingly.

Transition(*s*): Merge ⟶ No-Maneuver

Location: Controller

Trigger Event: TIMEOUT(TIME(Rcvd_merge_ok), 2*Sqrt(2*DistAhead/ACCEL_RATE))

Condition:

AND		OR	
Position **In State** Single	·	·	T
Position **In State** Leader	·	T	·
Position **In State** Not-Leader	T	·	·
Distance **In State** Between	·	T	T
Distance **In State** IAP	T	·	·
Motion **In State** Decelerate	T	T	T

Output Action:

Description: Platoon merges with platoon ahead.

Interface M_send_complete:

Destination: M_rcv_vehicle_complete

Trigger Event: Sendmerge-complete

Condition:

Output Action: SEND(Vehicle1_id)

Description: Send message to leader of platoon ahead, indicating completion of *Merge*.

4 Specification Analyses

The goal of our project is to explore the limits of automated analysis to provide information useful in safety-critical project development. Previously, Leveson and Stolzy described how backward analysis could be used to analyze a Time Petri-net model for safety [12], both with respect to the possibility of getting into hazardous states when the system operated as specified and when there were various types of failures.

Briefly, the procedure starts with a set of hazardous conditions. For each member of this set, the immediately prior state or states are generated from the inverse Petri net. Each of these "one-step-back" states is then examined to see if it is potentially a *critical state.* Informally, a critical state is defined as a state from which there is at least one path from which it is possible to reach a hazardous state (and possibly also non-hazardous states) and at least one path from which it is possible to reach only hazardous states. Identification of a critical state can be used to eliminate the path to the hazardous state or, if that is not possible, to design suitable controls. Note that it is necessary to look forward only one step from each potentially critical state in order to label it as critical (i.e., there exists a next state that is not hazardous). If it is not critical, it will be eliminated by the algorithm in a later state.

The procedure starts with partial states only. Some conditions in the state are unimportant as far as safety is concerned; therefore, the complete composition of the reachable hazardous states (i.e., the complete states from which to start the analysis) is not known at the beginning of the algorithm. The "don't-care" places in each state are filled in during the course of the analysis with the conditions that are possible given the particular model under consideration.

The procedure described considers only hazardous states that could be reached if the system operated correctly, i.e., it detects errors in the specification. Additional analysis procedures can be used to analyze the effects of faults and failures during operation of the system and thus to aid in the design of fault-tolerance and fail-safe mechanisms.

In the past, we developed algorithms to assist in performing some types of safety analyses on state-based specifications. We are now building tools to automate these algorithms and develop new types of safety analyses for requirements specifications written in RSML. We have also explored the application of our safety analysis procedures defined on Petri-nets to more complex RSML models and the automatic synthesis of fault trees from the model.

The core of our analysis tools is a simulator that is able to read RSML specifications. The simulator executes the specifications and assists in certain

analyses. Forward simulation of a specification provides the designer with information from which to determine if the specification conforms to the desired behavior. (At present, the simulator is run through a textual interface. A graphical user interface is under development.)

Automatic fault tree generation from the specification, based on backward simulation, allows a designer to see if, and how, a system can enter a hazardous state. In addition, Heimdahl [4] has developed algorithms and an automated tool to check an RSML specification for completeness and consistency. The tool also detects nondeterminism in the specification. Nondeterministic specifications may not only be unsafe, but they also make safety analysis less feasible. Although we do not require that detected nondeterministic behavior be removed from the specification, the instances that we detected in our TCAS II specification led to potentially unsafe behavior and basically reflected errors in the specification process.

Before describing the analysis tools, a definition of the term *configuration* as used in this paper is required. A *configuration* is a complete set of states in which the system can exist at some given moment. For example, the `Controller` system can be in a configuration where `Maneuver-Status` is in state `No-Maneuver`, `Distance` is in state `IP`, `Motion` is in state `Steady`, and `Position` is in state `Single`. This configuration reflects the situation of a vehicle on the automated highway moving at a steady speed, not performing any maneuver, and being the only vehicle in its platoon. The configuration is textually represented as *(MS:No_Maneuver, D:IP, M:Steady, P:Single)*. Similarly, a *partial configuration* is a configuration that specifies the states of only a subset of the components.

4.1 Forward simulation

Forward simulation can be started from a prespecified set of input messages and an initial system configuration. Simulation "steps" are divided into microsteps. A microstep is taken by choosing a set of transitions that are each triggered by an event generated during the previous microstep. This event may be generated by a transition, a message receipt, or a timeout. A full step is completed when no more microsteps can be taken. After completing a step, a system-wide queue is checked to determine when the next timeout or message is scheduled to occur. The global clock is advanced to this time, and the component that received the timeout or message begins a new step. The simulator can be executed from start to completion or it can be single-stepped (either a microstep or a step at a time), highlighting the currently active states on the screen.

Forward simulation allows for a check on the system specification to see whether it conforms to the way the system is supposed to work. It can also display whether the system, based on its specification, can get into undesired configurations.

The textual output from a forward simulation of the *Merge* maneuver is displayed in Figure 4. The graphical user interface will provide more readable output. The simulation process is as follows:

The simulator is started through a textual Tcl interface in line *1.* The AHS system is configured into its default states (lines 2-10)—*(MS:No_Maneuver, D:IP, M:Steady, P:Single).* This default configuration represents a vehicle on the automated highway moving at a steady speed, not performing any maneuver, and being the only vehicle in its platoon. In order to simulate *Merge* to completion, we set up messages that will be sent to the `Controller` component at the appropriate times. (Messages are set with the `rsml_addMessage` command, and its syntax is `rsml_addMessage` <destination component> <destination interface> <list of arguments of message> <time at which message should be sent>).

The first message (line *12)* is sent by the roadside infrastructure suggesting the vehicle begin its maneuver. The message is received by the input interface `Sys_M_start` in the component `Controller`. The first parameter (1) refers to the id of the recipient, while the second parameter *(20)* indicates the distance between the recipient and the platoon in front of it. The message is to be sent at a time value of *5* units (the system is initialized at time 0). The second message (line **14)** simulates a response from the leader of the platoon in front agreeing to *Merge.* The simulator acknowledges the correctness of the messages by adding them to its message queue (lines *13* and *15).*

We now run the system to completion (line 16) with the command `rsml_runFW`. The simulator considers the earliest message or timeout, which, in this case, is the message from the environment to begin *Merge.* The message is received by the vehicle (line 17), and it triggers the event `Systemmerge` (line *18)* according to specification. This event then triggers the transition in line *20,* which further triggers the sending of a message (line *21).* The system changes state because of the transition taken (lines *22-23).* The vehicle has now sent a message to the leader of the platoon in front, indicating its desire to merge with the latter's platoon. No more events are active and, hence, no more transitions are taken. The system thus completes a step (or a macrostep) (line **26).**

The simulator considers the next message or timeout, which happens to be the second message sent earlier. Receipt of this message enables the vehicle to continue with *Merge.* The vehicle begins to accelerate forward, and the system changes state because of three orthogonal transitions (lines *30-32).*

```
1:  curie% rsml_sim -i text_int.tcl
2:  Entering state "Controller No-Maneuver"
3:  Entering state "Controller Maneuver_Status"
4:  Entering state "Controller IP"
5:  Entering state "Controller Distance"
6:  Entering state "Controller Steady"
7:  Entering state "Controller Motion"
8:  Entering state "Controller Single"
9:  Entering state "Controller Position"
10: Entering state "Controller Controller_Machine"
11: AHS
12: rsml v2.9c (textual interface) $ rsml_addMessage Controller Sys_M_start {1 20} 5
13: RSML cb Adding Delta: message ({Controller Sys_M_start {1 20) 5)) 12
14: rsml v2.9c (textual interface) $ rsml_addMessage
    Controller M_rcv_ok {1 5} 10
15: RSML cb Adding Delta: message {(Controller M_rcv_ok {1 5) 10}} 13
16: rsml v2.9c (textual interface) $ rsml_runFW
17: message received: "Controller Sys_M_start {1 20) 5"
18: event "System_merge" triggered
19: ---------- End of a microstep ----------
20: transition "Controller No_Maneuver-to-Wait1-2" taken
21: message sent: "EXTERNAL EXTERNAL {1 0 0 20} 5 M_send"
22: Leaving state "Controller No_Maneuver"
23: Entering state "Controller Waitl"
24: event "Req_merge" triggered
25: ---------- End of a microstep ----------
26: ========== End of a macrostep ==========
27: message received: "Controller M_rcv_ok {1 5) 10"
28: event "Rcvd_merge_ok" triggered
29: ---------- End of a microstep ----------
30: transition "Controller Waitl-to-Merge" taken
31: transition "Controller IP-to-Between" taken
32: transition "Controller Steady-to-Accelerate" taken
33: Leaving state "Controller Waitl"
34: Entering state "Controller Merge"
35: Leaving state "Controller IP"
36: Entering state "Controller Between"
37: Leaving state "Controller Steady"
38: Entering state "Controller Accelerate"
39: ---------- End of a microstep ----------
40: ========== End of a macrostep ==========
41: event "TIMEOUT (TIME(PREV(0)Rcvd_merge_ok ),
    Sqrt(TWO * Dist_ahead / ACCEL-RATE))" triggered
42: ---------- End of a microstep ----------
```

```
43:  transition "Controller Accelerate-to-Decelerate" taken
44:  Leaving state "Controller Accelerate"
45:  Entering state "Controller Decelerate"
46:  ---------- End of a microstep ----------
47:  ========== End of a macrostep ==========
48:  event "TIMEOUT (TIME(PREV(0)Rcvd_merge_ok ),
     TWO * Sqrt(TWO * Dist_ahead / ACCEL-RATE))"  triggered
49:  ---------- End of a microstep ----------
50:  transition "Controller Merge-to-No-Maneuver" taken
51:  transition "Controller Between-to-IAP-2" taken
52:  transition "Controller Decelerate-to-Steady" taken
53:  transition "Controller Single-to-Not-Leader" taken
54:  message sent: "EXTERNAL EXTERNAL {1} 26 M_send_complete"
55:  Leaving state "Controller Merge"
56:  Entering state "Controller No-Maneuver"
57:  Leaving state "Controller Between"
58:  Entering state "Controller IAP"
59:  Leaving state "Controller Decelerate"
60:  Entering state "Controller Steady"
61:  Leaving state "Controller Single"
62:  Entering state "Controller Not-Leader"
63:  event "Send-merge-complete" triggered
64:  ---------- End of a microstep ----------
65:  ========== End of a macrostep ====×=====
66:  ########## End of simulation ############
```

Figure **4:** *Forward Simulation Example (contd.)*

The event `Rcvd_merge_ok`, which was triggered because of the receipt of the second message (line 28), now causes a timeout event to be generated (line 41). This timeout event is an indication for the vehicle to begin decelerating (line 43). Another timeout event at a later time causes the vehicle to change its speed to a steady speed (line 48). This second timeout event also indicates the end of the maneuver, and the vehicle moves into its new configuration indicating that it is now part of a platoon (lines 50-53). The vehicle also sends a message to the leader of its platoon (line 54) indicating that it has completed the maneuver.

4.2 Fault Tree generation

Fault Tree Analysis (FTA) is a form of safety analysis widely used in the aerospace, electronics, and nuclear industries. The technique was originally developed in 1961 at Bell Labs to evaluate the Minuteman Launch Control System for an unauthorized missile launch.

The top event in a fault tree is a hazardous condition or state of the system, where a hazard is defined as a state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object), will lead to an accident (loss event) [8]. FTA uses Boolean logic to describe the combinations of events and states that constitute a hazardous state. Each level in the tree lists the events and states that are necessary to cause or lead to the state shown in the level above it.

Previously, Leveson and colleagues explored the generation of fault trees from code [10, 9]. Basically, generating code-level fault trees assumes that system-level fault trees have already been built down to the software interface. In contrast, this paper examines the practicality of producing *system-level* (rather than code-level) fault trees from a state-machine model. Because producing fault trees is labor-intensive and error-prone and depends on the analyst's understanding of the operation of the system, attempts have been made to synthesize these trees automatically. Several procedures for automatic synthesis have been proposed, but these work only for systems consisting purely of hardware elements.

In the automated approaches, a model of the hardware, such as a circuit diagram, is used to generate the tree [1, 2, 7]. Taylor's technique, which is typical, takes the components of the hardware model and writes them as transfer statements [15]. Each statement describes how an output event from the component can result from the combination of an internal change in the component and an input event and how the component state changes in response to input events. In general, the transfer statement will be conditional on the previous

component state. Together, the transfer statements form the transfer function for the component.

Both the normal and failure properties of the component are described, and each transfer statement is represented as a small fragment of a fault tree that Taylor calls a *mini-fault tree.* The synthesis process consists of building the fault tree by matching the inputs and outputs of these mini-fault trees.

To generate more general fault trees, a model that includes more than just hardware circuits is required. We are exploring techniques to generate fault trees from state-machine models.

Automatic fault tree generation from an RSML specification is based on a backward simulation of the system. Backward simulation involves finding configurations such that there exist a set of transitions that can lead from each of these configurations to the current configuration in a microstep. For each such "one-step-back" configuration found, fault tree templates (representing mini-fault-trees) are created. These templates contain detailed information on how the system could move from the one-step-back configuration to the current configuration. Thus the backward reachability tree provides the basic structure on which the fault tree is built.

Backward execution can be used to implement the safety analysis algorithm described earlier and originally defined for Time Petri-nets. A particular starting configuration can have zero or more one-step-back configurations, such that a set of parallel transitions can cause the system to move from each one-step-back configuration to the starting configuration in a microstep. For every one-step-back configuration constructed, the algorithm considers those configurations that can be reached in one forward microstep. The information obtained can be used to eliminate paths to hazardous states from the model.

Backward simulation can currently be performed one microstep at a time. Hence fault trees are automatically constructed one backward microstep at a time. Larger fault trees can easily be built by repeating the symbolic backward simulation, starting from an appropriate one-step-back configuration each time. Once the backward simulation has been repeated a desired number of backward steps, the entire fault tree can be generated. By generating the tree one step at a time, we allow the possibility of having a human analyst prune the tree of physically impossible branches to save time and effort.

We first describe the templates used in creating fault trees. We then demonstrate the fault tree procedure with the help of an example. Typically, a great many one-step-back configurations are possible for an initial configuration. The fault tree generation tool can eliminate impossible one-step-back configurations based on IN-STATE conditions. This pruning technique is described next. Finally, we show how fault tree analysis can be used to modify the

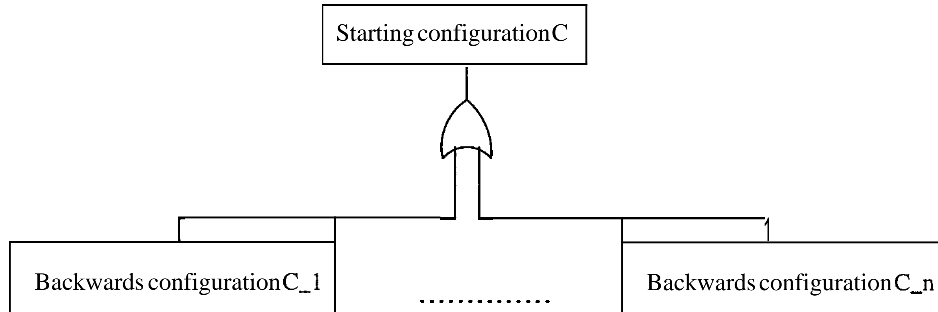


Figure 5: *Basic template for each level of fault tree.*

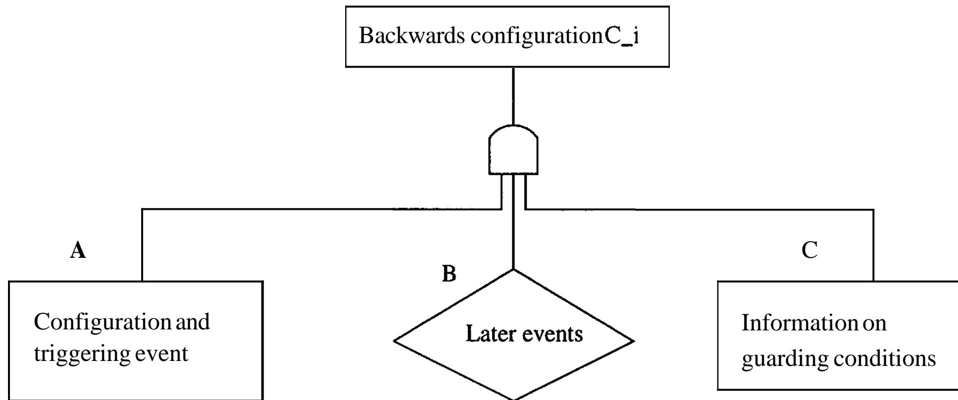


Figure 6: *Expansion of a backward configuration node.*

system design so that it can handle failures.

4.2.1 Fault tree templates

The fault tree generator constructs mini-fault-trees, each of which represents a path from a configuration to one of its one-step-back configurations. These mini-fault-trees are constructed from a set of templates that describe in greater detail how the system could move from one configuration to another. As noted earlier, Taylor uses a similar notion of mini-fault-trees to represent small

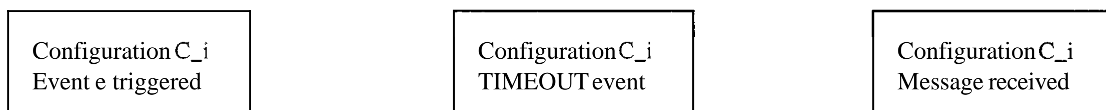


Figure 7: *Choices of triggering event nodes.*

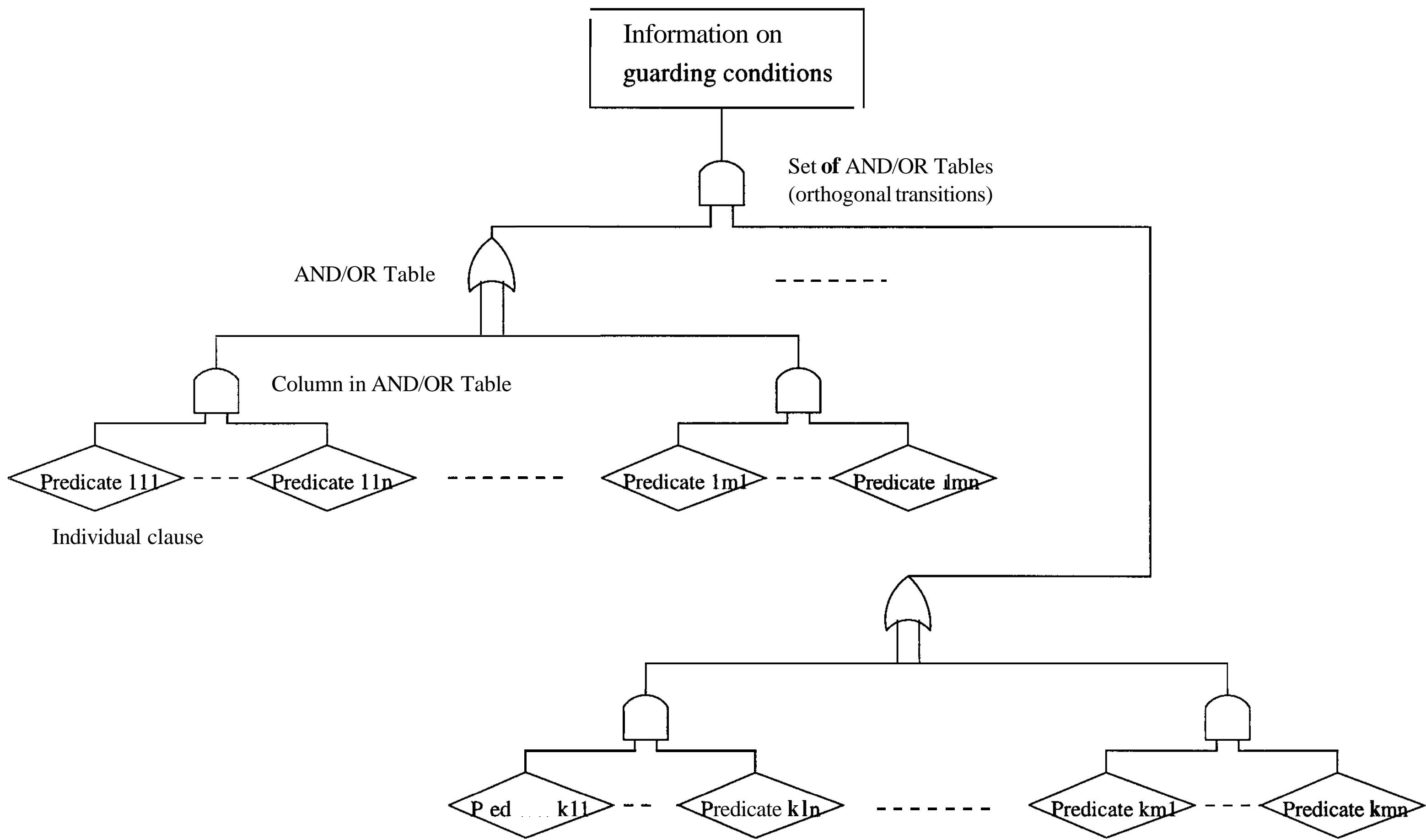


Figure 8: *Expansion of guarding conditions information node.*

fragments of fault trees that are generated from hardware circuit diagrams [15].

Given an initial configuration, there can be zero or more one-step-back configurations, such that there exists a set of orthogonal transitions that can cause the system to move from each of these backward configurations to the initial configuration in a microstep. Figure 5 shows the template corresponding to this situation. Figure 6 displays the expansion of each backward configuration node. The orthogonal transitions are triggered by a set of simultaneous events (there may be one event in this set that triggers all the orthogonal transitions, or there may be more than one). This situation is depicted by node A. Furthermore, the events in this set may need to be triggered before other events in order for the system to move to the starting configuration. This information is shown in node B. Finally, the guarding condition(s) on each of the orthogonal transitions, if any, need(s) to be satisfied in order for the transitions to be valid (node C). An event can be generated as an action of a transition, as a result of a message received, or as a result of a timeout. Figure 7 displays these choices. Finally, Figure 8 shows a template that represents the set of guarding conditions that are to be satisfied, in terms of each individual guarding condition. This template depicts an expansion of node C in Figure 6.

Perhaps the easiest way to understand the procedure is to look at an example.

4.2.2 Fault tree example

For the example, we assume a safety constraint for the AHS that platoons must be at least an inter-platoon distance apart from each other. This inter-platoon distance is such that a vehicle decelerating at full platoon breaking can avoid colliding with the vehicle ahead when the latter is decelerating at some standard greater rate. A hazard arises when this condition is violated, i.e., when the leader of a platoon is at less than an inter-platoon distance behind a vehicle. Such a situation can be represented by the partial configuration C, *(D:Between, P:Leader).* The fault tree, with C as root, is shown in Figure 9. (The RSML fault tree generator outputs the fault tree in a file using a format that is readable by *dotty,* a graph layout program that generates and displays graphs. The fault tree generator can be easily modified to output the tree in some other format if required.)

The fault tree displays three sub-trees that can lead to C: one corresponding to *Merge,* one to *Split* where the vehicle is just behind the leader of the platoon and the latter decides to split, and one where the vehicle is not the leader of a platoon and decides to cause a split in the platoon. The subtrees display how the system can end up in the configuration C (the events, mes-

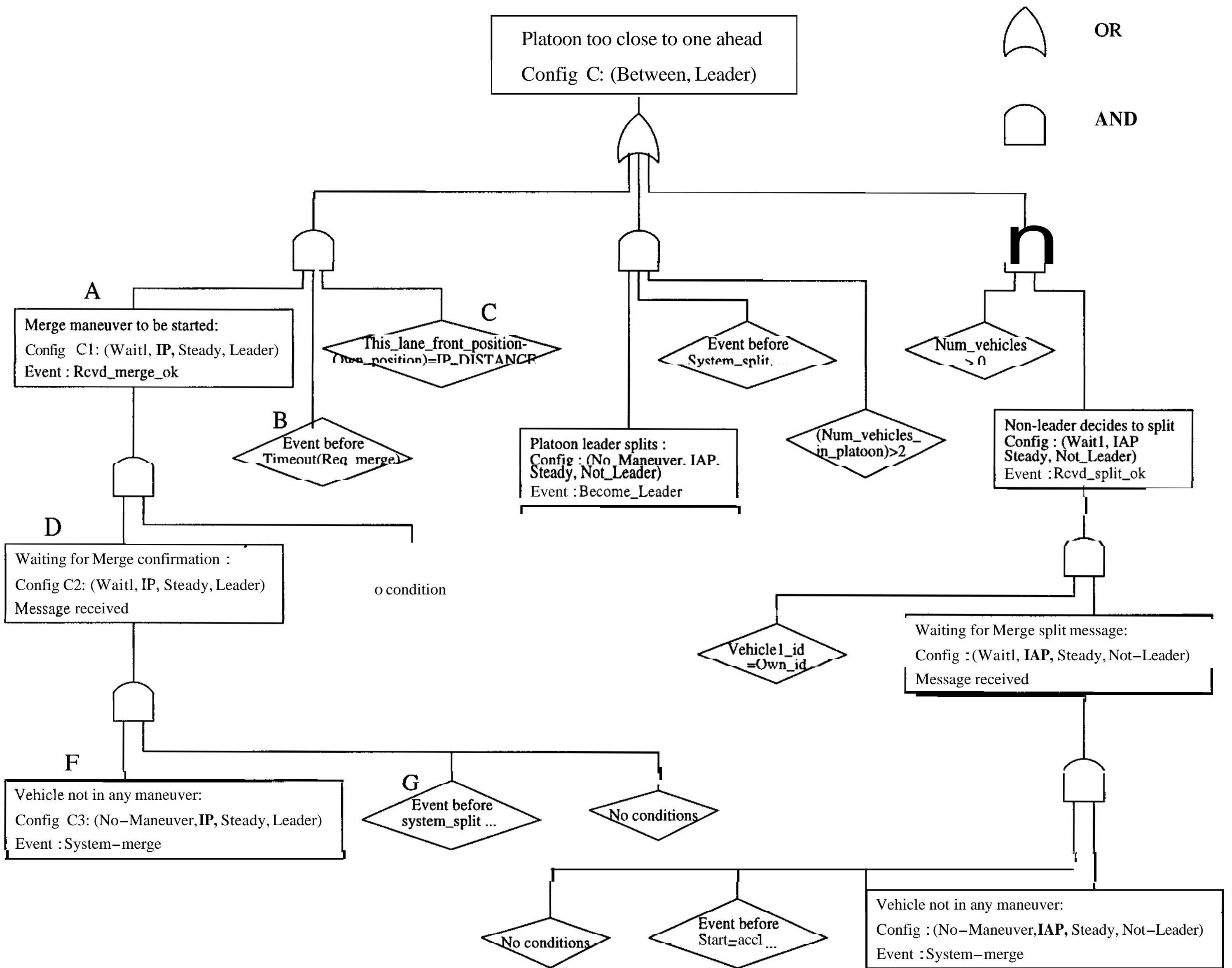


Figure 9: *An example of an AHS fault tree generated from an RSML specification.*

sages, and conditions required) starting from the configurations at the lowest level.

Consider the leftmost subtree of C. One backward configuration from C was found. This configuration, C_1, *(MS:Wait1, D:IP, M:Steady, P:Leader)*, is represented by node **A** in Figure 9 and denotes the situation where the leader of a platoon is waiting for a response from a vehicle. The set of transitions that take the system from C_1 to C are triggered by the event Rcvdmerge-ok. This set consists of the following orthogonal transitions:

- The transition [Wait1 ⟶ Merge] (in the state-machine Maneuver-Status),
- The transition [IP ⟶ Between] (in the state-machine Distance), and
- The transition [Steady ⟶ Accelerate] (in the state-machine Motion).

Unlike C, C_1 is not a partial configuration and requires that Maneuver-Status and Motion be in specific states. Such a requirement is a result of the IN-STATE guarding conditions on the three transitions listed above.

Node B in the fault tree specifies that the event Rcvd_merge_ok must be triggered before a timeout event on Reqmerge moves the system from C_1 to C (this timeout event causes the system to abort *Merge* if the leader of the platoon ahead does not respond to a *Merge* request within an appropriate amount of time). For every one-step-back configuration constructed, the fault tree generator considers those configurations that can be reached in one *forward* microstep. If these one-step-forward configurations exist, a set of events triggering transitions to each such configuration is constructed and displayed. This analysis is based on the critical state analysis for Petri nets that was described in the previous section. Such information can be useful in identifying situations where a missed event or transition could cause the system to get into a hazardous situation. Finally, node C requires that the condition "This-lane-front-position – Own-position = IP-DISTANCE,' needs to be true for the transitions to be taken.

The next level of the tree (the sub-tree with node **A** as root) describes how the system can get into C_1. One backward reachable configuration, call it C_2, was found (node D). C_2 represents the situation where the Controller component receives a message that causes the event Rcvd_merge_ok to be triggered, which in turn triggers the transitions connecting C_1 and C. Node E indicates that there are no conditions on the receipt of the message and the triggering of Rcvd_merge_ok. The state machines remain in the same states while the message is received.

At the next level, node F represents the configuration C_3, *(MS:No_Maneuver, D:IP, M:Steady, P:Leader)*. C_3 represents the configuration of a vehicle at

the beginning of *Merge.* The system can move from C_3 to C_2 if the event Systemmerge is triggered before Rcvd_req_chng_lane, System-split, or Rcvd_req_split (node G).

In summary, the example fault tree generated by the tool shows that a vehicle can end up in a potentially hazardous situation (denoted by the configuration C) in three different ways (represented by the three sub-trees of the root node of the fault tree), based on the specification describing the normal functioning of the system. If *Merge* or *Split* were completed, it would move from C to a safe configuration according to the specification (for example, in *Merge,* the vehicle would slow down appropriately to merge with the one in front and not crash into it). The fault tree does reveal that if the system was in C and some failure occurred (a communication error, for example) that prevented it from continuing according to its specified behavior, a collision could result. Hence the fault tree identifies situations where adequate care needs to be taken to ensure correct behavior or risk-minimization mechanisms need to be added to prevent a catastrophe in the presence of failures. Section **4.3** describes another such situation and suggests a way to modify the design to incorporate a fault-handling mechanism.

4.2.3 IN-STATE **Pruning**

Typically, a great many backward configurations are possible from an initial configuration. However it may be impossible for the system to move from some of these one-step-back configurations to the initial configuration because the guarding conditions on the set of orthogonal transitions cannot all be satisfied. The fault tree generation tool can eliminate impossible one-step-back configurations based on IN-STATE guarding conditions. (A similar sort of pruning can be performed for IN-ONE-OF guarding conditions also.) To explain this pruning technique, consider an example. Suppose a vehicle is in configuration C_1, where C_1 is represented by *(MS:No_Maneuver, D:IP, M:Accelerate, P:Single),* i.e., it is a single vehicle and accelerating. In the AHS specification, there exists a transition T, [Steady   Accelerate], triggered by the event Rcvd_merge_ok. So the configuration C_2, *(MS:No_Maneuver, D:IP, M:Steady, P:Single)*, is a possible one-step-back configuration of C_1. However, one of the IN-STATE guarding conditions on transition T is "Maneuver–Status In–State Waitl", which is not satisfied if the system were in C_2. Hence, C_2 fails to be a valid backward configuration for C_1 and can be eliminated from the fault tree for C_1. Such pruning can drastically cut the number of one-step-back configurations. Manual pruning can also be performed by the analyst. We are exploring additional ways to provide automatic pruning.

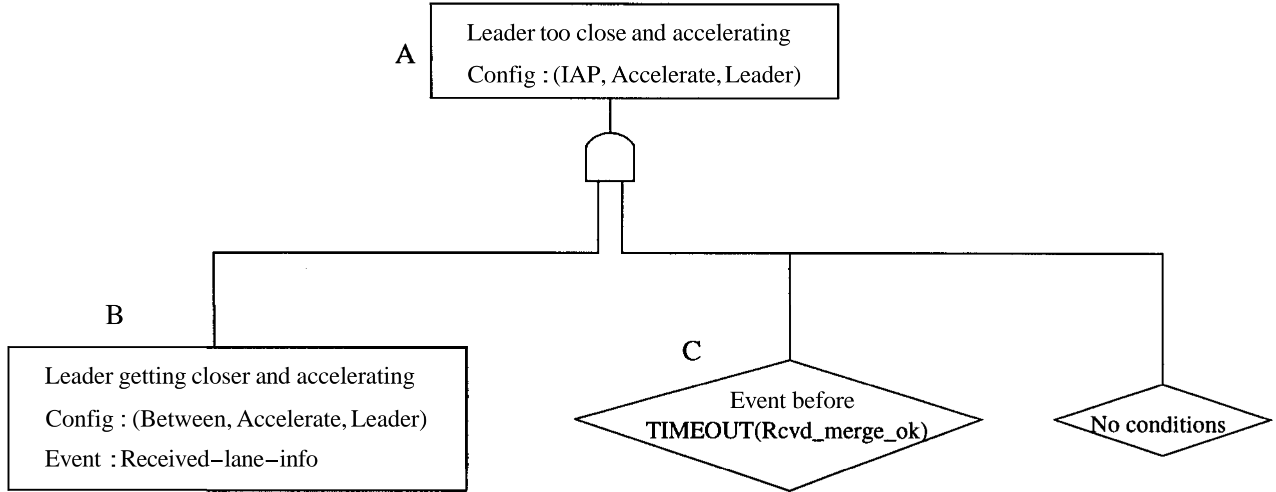


Figure 10: *Fault tree example.*

4.3 Handling failures

Consider the hazardous situation where the leader of a platoon is very close to the vehicle in front of it (at an intra-platoon distance) and accelerating. Such a situation can be represented by the partial configuration C defined as *(M:Accelerate, D:IAP, P:Leader).* Figure 10 displays the fault tree with C as its root (node **A**). Only one one-step-back configuration (C_1), defined as *(M:Accelerate, D:Between, P:Leader)* and represented by node B, was found by the fault tree generator. C_1 represents the situation where the leader is still accelerating, but it is farther away from the vehicle in front (the state machine `Distance` is in state `Between`). The system moves from C_1 to C when the event `Received-lane-info` is generated (node B). This event is generated by the receipt of a message from the vehicle's forward sensor indicating its distance from the vehicle in front of it. The state machine `Distance` changes state through the transition `[Between` $\longrightarrow$ `IAP]`, reflecting the sensor input. Node C shows that the system can move from C_1 to C provided that the sensor input is received before a timeout event based on `Rcvd_merge_ok`. In terms of the RSML specification, this timeout event triggers the transition `[Accelerate` $\longrightarrow$ `Decelerate]` during *Merge.* The timeout in effect slows the vehicle down at the appropriate instant so that it merges with the platoon ahead without crashing into it. Within the confines of the normal behavior of the system, this timeout event and the associated transition would occur before the transition `[Between` $\longrightarrow$ `IAP]` that causes the system to move from C_1 to C. However, node C reveals that if either the timeout event or its related

transition fail (because of a faulty communications component or a faulty deceleration device, for example), the system can move into the hazardous situation represented by C, thus potentially causing an accident.

Information from the fault tree can thus be used to identify safety-critical areas and situations where a failure in the system (for example, a failure that leads to the transition [Accelerate ⟶ Decelerate] in *Merge* being missed) can place the system in a hazardous configuration. The design of the AHS can be then be strengthened appropriately to prevent such a configuration from occurring. Leveson and Stolzy ([12]) show how interlocks can be incorporated into Petri net models to ensure that a desirable transition has precedence over an undesirable one. In the AHS model, we would like the transition [Accelerate ⟶ Decelerate] to be taken before the transition [Between ⟶ IAP]. If the desired former transition does not get taken, we would like the system to act appropriately to prevent the vehicle from entering the state IAP while still accelerating.

One way the AHS specification can be modified to prevent the system from moving into configuration C in the presence of a transition failure (failing to decelerate) is to introduce a watchdog timer. This timer can be treated as another vehicle component that can communicate with the controller. The timer is set by a message from the controller, and after a specified period of time, it sends a message back to the controller. The timer can be used to set a limit on how long a vehicle can accelerate. Thus, if a vehicle is in a situation described above where it fails to decelerate while participating in *Merge* or *Split,* the timer or controller can initiate a backup deceleration mechanism. In order for this to work, the timer should be set for a time period (1) greater than the time the vehicle would normally accelerate as part of *Merge* or *Split* and (2) less than the time it would take for the vehicle to collide with the one in front if it continued accelerating. If the vehicle decelerates normally, the message from the timer is ignored. However, if the vehicle fails to decelerate normally, the message from the timer can be used to place the vehicle in a special configuration where it can decelerate at a higher rate than normal (emergency braking) to avoid collision.

We modified the AHS design to incorporate the timer, adding both transitions and interfaces. An extra state, Decelerate-Quick, was added to represent the state where the vehicle is decelerating quicker than normal in order to slow down and avoid a collision. Figure 11 shows part of a forward simulation with the modified AHS specification. Lines 1-3 show the system in the middle of *Merge,* where the vehicle begins accelerating forward. At the same time, a message is sent to the timer (line 4) indicating the beginning of acceleration. When the vehicle fails to decelerate, i.e. it continues to be in its accelerated

```
     ...
 1:  transition "Controller Waitl-to-Merge" taken
 2:  transition "Controller IP-to-Between" taken
 3:  transition "Controller Steady-to-Accelerate" taken
 4:  message sent: "EXTERNAL EXTERNAL {10} 10 0Interface_send_timer"
     ...
 5:  message received: "Controller 1Interface_get_timer {} 20"
 6:  event "Timer-received" triggered
 7:  ---------- End of a microstep ----------
 8:  transition "Controller Accelerate-to-Decelerate_Quick" taken
 9:  Leaving state "Controller Accelerate"
10:  Entering state "Controller Decelerate-Quick"
11:  ---------- End of a microstep ----------
12:  ========== End of a macrostep ==========
13:  event "TIMEOUT (TIME(PREV(0)Timer_received ),
     Sqrt(TWO * Dist_ahead / QUICK-DECEL-RATE))" triggered
14:  ---------- End of a microstep ----------
15:  transition "Controller Merge-to-No-Maneuver" taken
16:  transition "Controller Between-to-IAP-2" taken
17:  transition "Controller Decelerate-Quick-to-Steady" taken
18:  transition "Controller Single-to-Not-Leader" taken
19:  message sent: "EXTERNAL EXTERNAL {1} 25 M_send_complete"
     ...
20:  ---------- End of a microstep ----------
21:  ========== End of a macrostep ==========
22:  ########## End of simulation ############
```

Figure 11: *Forward Simulation with timer.*

state, a message is received from the timer (line 5). The message causes the system to move into the special state Decelerate-Quick (line 8). After decelerating for a certain time (the quicker deceleration rate is determined by the constant QUICK-DECEL-RATE), the vehicle completes *Merge* in a normal and safe fashion (lines 15-19).

We have generated fault trees for other hazardous situations. For example, the AHS system can find itself in the hazardous configuration *(D:IAP, P:Between)* during a *Merge* or *Split* maneuver. We have also found that the specification prevents the system from entering configurations where the vehicle is not in the midst of any maneuver, yet its other state machines are not in their default states. An example of such a configuration is *(MS:No_Maneuver, M:Accelerate, P:Leader)*.

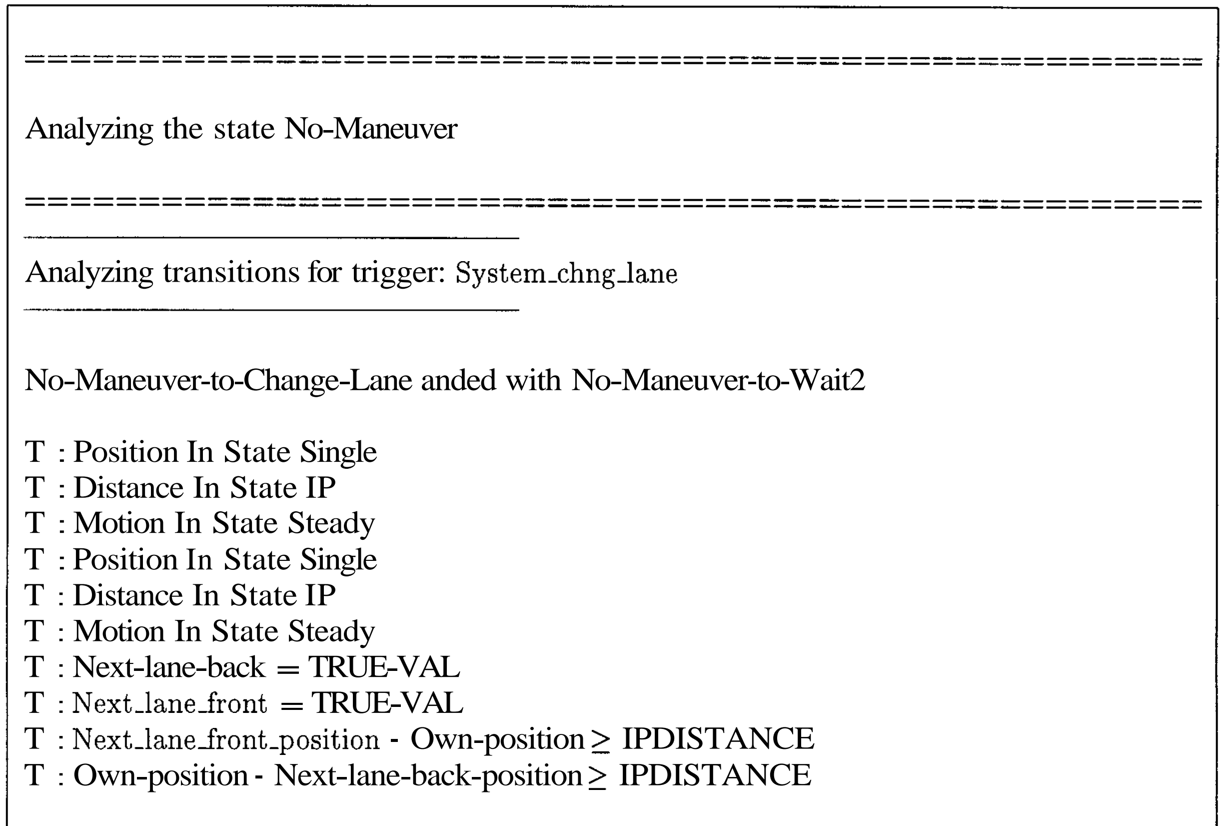
The fault tree analysis thus enables the designer to detect situations where failures can lead to accidents. Failure-handling mechanisms can then be added to make the system more resilient to failures.

4.4 Consistency analysis

Tools have also been developed to check the consistency and completeness of RSML specifications [4].

Consistency and logical completeness analysis on an RSML specification is performed automatically, and the analysis results are output to a file without user intervention. The output lists conditions on the transitions out of a state that allow more than one transition to be satisfied simultaneously or missing cases. Performing this analysis on our AHS model, two nondeterministic situations were detected, both arising during the beginning of *Change-lane*. The output from the analysis that checks for one of these non-deterministic situations is shown in Figure 12.

As described earlier, there are three cases for *Change-lane,* depending on how many vehicles are present in an adjacent lane within a critical distance: none, one, or at least two. With respect to the vehicle desiring to change lanes, a transition from `No_Maneuver` to one of Change-Lane, `Wait1`, or `Wait2`, respectively, can be taken, each triggered by the same event. In the AHS specification, the Boolean variables Next-lane-front and Next-lane-back indicate the presence of a vehicle in the adjacent lane within a critical distance in front of or behind the vehicle. These variables allow the system to determine which of the three transitions from No-Maneuver to take. For example, the transition [`No_Maneuver` ⟶ Change-Lane] is governed by the following two guarding conditions :



==

Analyzing the state No-Maneuver

==

Analyzing transitions for trigger: System_chng_lane

No-Maneuver-to-Change-Lane anded with No-Maneuver-to-Wait2

T : Position In State Single
T : Distance In State IP
T : Motion In State Steady
T : Position In State Single
T : Distance In State IP
T : Motion In State Steady
T : Next-lane-back = TRUE-VAL
T : Next_lane_front = TRUE-VAL
T : Next_lane_front_position - Own-position ≥ IPDISTANCE
T : Own-position - Next-lane-back-position ≥ IPDISTANCE

Figure 12: *Consistency analysis.*

- the expression Next-lane-front = TRUE-VAL is false (i.e. there is no vehicle ahead in the adjacent lane), and
- the expression Next-lane-back = TRUE-VAL is also false.

These two conditions were incorrectly left out of the guarding conditions for the transition. This omission means that if a vehicle is moving at a steady speed and is the only vehicle in its platoon, and either of Next-lane-front or Next-lane-back were true, then the transition from NoManeuver to Change-Lane can be taken, along with either of the transitions from NoManeuver to Wait1 or Wait2. This is an obvious error in the specification and can lead to a hazardous situation (For example, a collision can result if the first transition is taken and there is a vehicle in the adjacent lane).

The consistency and logical completeness analysis can thus greatly help in developing consistent, unambiguous specifications.

5 Conclusions

This paper describes tools for the safety analyses of RSML specifications. These tools include a forward simulator, a fault tree generator based on backward simulation, and a consistency and logical completeness checker. The automated analyses were demonstrated for a specific AHS model.

References

[1] David J. Allen. Digraphs and fault trees. *Hazard Prevention,* pages 22–25, January/February 1983.

[2] [ALM80] P.K. Andow, F.P. Lees, and C.P. Murphy. The propagation of faults in process plants: A state of the art review. In *7th International Symposium on Chemical Process Hazards,* pages 225–237. University of Manchester, Institute of Science and Technology, United Kingdom, April 1980.

[3] David Harel. Statecharts: A visual formalism for complex systems. *Science* of *Computer Programming,* 8:231-274, 1987.

[4] Mats P.E. Heimdahl and Nancy G. Leveson. Completeness and consistency checking of software requirements. In *Proceedings* of *the International Conference on Software Engineering,* IEEE Computer Society, Los Alamitos, Calif., April 1995.

[5] A. Hitchcock. A specification of an automated freeway with vehicle-borne intelligence. PATH Research Report, University of California, Berkeley, 1992.

[6] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya. The Design of Platoon Maneuver Protocols for AHS. PATH Research Report UCB-ITS-PRR-91-6. University of California, Berkeley, CA., 1991.

[7] Frank P. Lees. *Loss Prevention in the Process Industries, Vol. 1 and 2.* Butterworths, London, 1980.

[8] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Co., 1995

[9] Nancy G. Leveson, Stephen S. Cha, and Timothy J. Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Software,* 8(7):48–59, July 1991.

[10] Nancy G. Leveson and Peter R. Harvey. Analyzing software safety. *IEEE Transations on Software Engineering,* SE-9(5):569–579, September 1983.

[11] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering,* SE-20(9), September 1994.

[12] Nancy G. Leveson and Janice L. Stolzy. Safety analysis using Petri nets. *IEEE Transations on Software Engineering,* SE-13(3):386–397, March 1987.

[13] Robyn R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. *IEEE Software Requirements Conference,* San Diego, January 1992.

[14] Jon Damon Reese. *Software Deviation Analysis.* embedded systems. Ph.D. Dissertation, University of California, Irvine, 1995.

[15] J.R. Taylor. An integrated approach to the treatment of design and specification errors in electronic systems and software. In E. Lauger and J. Moltoft, editors, *Electronic Components and Systems,* North-Holland Publishing Co., 1982.

A Appendix A: Vehicle-Controller Example Specification

This appendix includes an example specification of a vehicle controller component. This model is intended only as an example and does not represent any real AHS design.

A.1 Interface

This section defines the interfaces between the controller component and the components with which it communicates.

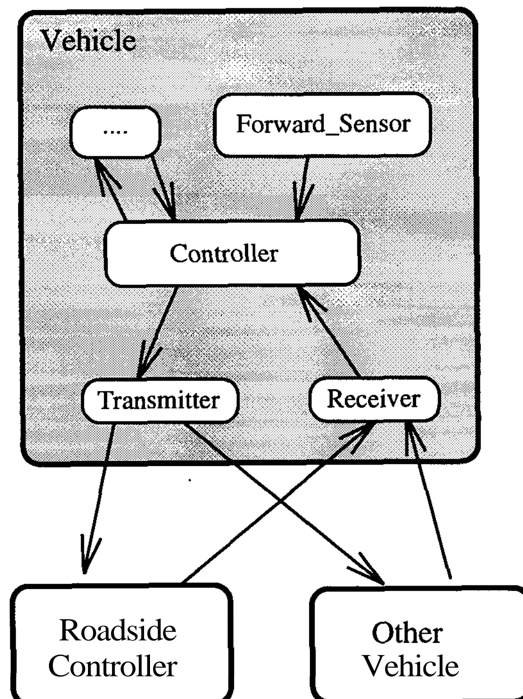


Figure 13: *The components and interface messages. Only the messages to and from the Vehicle Controller are included here. A complete system specification would include all the messages.*

Input interfaces

The model has 17 different input messages to the vehicle controller as described below.

Interface Sys_CL_start:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE(Vehicle1_id$_{\text{v-80}}$, Vehicle2_id$_{\text{v-81}}$)
Condition:
Output Action: System_chng_lane$_{\text{e-84}}$
Description: Message to begin Change-Lane maneuver.

Interface CL_rcv:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE(Vehicle1_id$_{\text{v-80}}$, Vehicle2_id$_{\text{v-81}}$)
Condition:

AND	
Vehicle1_id$_{\text{v-80}}$ = Own_id$_{\text{v-80}}$	T

Output Action: Rcvd_req_chng_lane$_{\text{e-84}}$
Description: Message from vehicle in adjacent lane that wants to change lanes. From CL_send1 or CL_send2.

Interface CL_rcv_vehicle_complete:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE(Vehicle1_id$_{\text{v-80}}$)
Condition:

AND	
Vehicle1_id$_{\text{v-80}}$ = Own_id$_{\text{v-80}}$	T

Output Action: Vehicle_chng_lane_complete$_{\text{e-84}}$
Description: Message indicating another vehicle has completed Change-lane maneuver. From CL_send_complete1.

Interface CL_rcv_ok:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE(Vehicle1_id$_{v-80}$)
Condition:

AND		
	Vehicle1_id$_{v-80}$ = Own_id$_{v-80}$	T

Output Action: Rcvd_chng_lane_ok$_{e-84}$
Description: Indication from vehicle in adjacent lane that it is OK to continue Change-lane maneuver. From CL_send_ok.

Interface Sys_M_start:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE(Vehicle1_id$_{v-80}$, Dist_ahead$_{v-81}$)
Condition:
Output Action: System_merge$_{e-84}$
Description: Message to begin Merge maneuver.

Interface M_rcv:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE(Vehicle1_id$_{v-80}$, Vehicle2_id$_{v-81}$, Num_vehicles$_{v-80}$, Dist_ahead$_{v-81}$)
Condition:

AND		
	Vehicle1_id$_{v-80}$ = Own_id$_{v-80}$	T

Output Action: Rcvd_req_merge$_{e-84}$
Description: Message from leader of rear platoon that wants to merge. From Msend.

Interface M_rcv_ok:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($Vehicle1_id_{v\text{-}80}$, $Num_vehicles_{v\text{-}80}$)
Condition:
Output Action: $Rcvd_merge_ok_{e\text{-}84}$
Description: Message from leader of platoon in front indicating Merge is OK. From M_send_ok.

Interface M_rcv_vehicle_complete:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($Vehicle1_id_{v\text{-}80}$)
Condition:

AND		
	$Vehicle1_id_{v\text{-}80} = Own_id_{v\text{-}80}$	T

Output Action: $Vehicle_merge_complete_{e\text{-}84}$
Description: Message from leader of merging platoon indicating its platoon has merged. From M_send_complete.

Interface Sys_S_start:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($Vehicle1_id_{v\text{-}80}$, $Num_vehicles_{v\text{-}80}$)
Condition:
Output Action: $System_split_{e\text{-}84}$
Description: Message to begin Split maneuver.

Interface S_rcv_back:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($\text{Vehicle1_id}_{\text{v-80}}$)
Condition:

AND | $\text{Vehicle1_id}_{\text{v-80}} = \text{Own_id}_{\text{v-80}}$ | ☐

Output Action: $\text{Become_leader}_{\text{e-84}}$
Description: Message from leader of platoon that is splitting away. From S_send_back.

Interface S_rcv_complete1:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($\text{Vehicle1_id}_{\text{v-80}}$)
Condition:

AND | $\text{Vehicle1_id}_{\text{v-80}} = \text{Own_id}_{\text{v-80}}$ | ☐

Output Action: $\text{Vehicle_split_complete}_{\text{e-84}}$
Description: Message from your leader that split, indicating it has completed its Split maneuver. From S_send_complete1.

Interface S_rcv_leader:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($\text{Vehicle1_id}_{\text{v-80}}$, $\text{Vehicle2_id}_{\text{v-81}}$, $\text{Num_vehicles}_{\text{v-80}}$)
Condition:

AND | $\text{Vehicle1_id}_{\text{v-80}} = \text{Own_id}_{\text{v-80}}$ | ☐

Output Action: $\text{Rcvd_req_split}_{\text{e-84}}$
Description: Message to leader from vehicle in platoon that wants to split. From S_send_leader.

Interface S_rcv_all:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($Vehicle1_id_{v\text{-}80}$)
Condition:

AND	
	$Vehicle1_id_{v\text{-}80} = Own_id_{v\text{-}80}$

Output Action: $Start_accl_{e\text{-}84}$
Description: Message from leader of platoon informing about split in platoon. From S_send_all.

Interface S_rcv_splitter:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($Vehicle1_id_{v\text{-}80}$)
Condition:

AND	
	$Vehicle1_id_{v\text{-}80} = Own_id_{v\text{-}80}$	T

Output Action: $Rcvd_split_ok_{e\text{-}84}$
Description: Message from leader of platoon to vehicle that wants to split. From Ssendsplitter.

Interface S_rcv_complete2:

Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE($Vehicle1_id_{v\text{-}80}$)
Condition:

AND	
	$Vehicle1_id_{v\text{-}80} = Own_id_{v\text{-}80}$	T

Output Action: $Vehicle_split_complete_{e\text{-}84}$
Description: Message from your leader that split, along with other vehicles that were in front of you. From S_send_complete2.

Interface Rcv_own_info:
Source: Forward Sensor
Destination: Vehicle Controller
Trigger Event: RECEIVE(Dist_ahead$_{v\text{-}81}$)
Condition:
Output Action: Received_lane_info$_{e\text{-}84}$
Description: Message from forward sensor indicating distance of vehicle ahead.

Interface Get-positions:
Source: Receiver
Destination: Vehicle Controller
Trigger Event: RECEIVE(Own_position$_{v\text{-}80}$ This_lane_front_position$_{v\text{-}79}$)
Condition:
Output Action:
Description: Receive information on parameters.

Output Interfaces

The vehicle controller can transmit **14** different output message types.

Interface CL_send1:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Req_chng_lane1 $_{e\text{-}84}$
Output Action: SEND(Vehicle1_id$_{v\text{-}80}$, Own_id$_{v\text{-}80}$)
Description: Message to leader of platoon in adjoining lane, indicating desire to change lanes.

Interface CL_send2:

Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Req_chng_lane2 $_{e\text{-}84}$
Output Action: SEND(Vehicle1_id$_{v\text{-}80}$, Own_id$_{v\text{-}80}$)
Description: Message to leader of other platoon in adjoining lane, indicating desire to change lanes.

Interface CL_send_ok:

Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Chng_lane_ok $_{e\text{-}84}$
Output Action: SEND(Vehicle2_id$_{v\text{-}81}$)
Description: Message to vehicle that wants to change lane, indicating approval of Change-lane maneuver.

Interface CL_send_complete1:

Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Change_lane_complete $_{e\text{-}84}$
Output Action: SEND(Vehicle1_id$_{v\text{-}80}$)
Description: Message to leader of platoon in adjacent lane, indicating completion of Change-lane maneuver.

Interface CL_send_complete2:

Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Change_lane_complete $_{e\text{-}84}$
Output Action: SEND(Vehicle2_id$_{v\text{-}81}$)
Description: Message to leader of other platoon in adjacent lane, indicating completion of Change-lane maneuver.

Interface M_send:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Req_merge $_{e\text{-}84}$
output Action: SEND(Vehicle1_id$_{v\text{-}80}$, Own_id$_{v\text{-}80}$, Num_vehicles_in_platoon$_{v\text{-}80}$, Dist_ahead$_{v\text{-}81}$)
Description: Message to leader of platoon ahead, indicating desire to merge.

Interface M_send_ok:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Merge-ok $_{e\text{-}84}$
Output Action: SEND(Vehicle2_id$_{v\text{-}81}$, Num_vehicles_in_platoon$_{v\text{-}80}$, Dist_ahead$_{v\text{-}81}$)
Description: Message to leader of rear platoon that wants to merge, indicating approval.

Interface M_send_complete:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Sendmerge-complete $_{e\text{-}84}$
Output Action: SEND(Vehicle1_id$_{v\text{-}80}$)
Description: Message to leader of platoon ahead, indicating completion of Merge maneuver.

Interface S_send_back:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Send_leader_split $_{e\text{-}84}$
Output Action: SEND(Vehicle1_id$_{v\text{-}80}$)
Description: Message from leader of platoon that wants to split to vehicle behind.

Interface S_send_complete1:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Send_spl_complete1 $_{e\text{-}84}$
Output Action: SEND(Vehicle1_id$_{v\text{-}80}$)
Description: Message to new leader of platoon behind, indicating the completion of Split maneuver.

Interface S_send_leader:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Reqsplit $_{e\text{-}84}$
Output Action: SEND(Vehicle1_id$_{v\text{-}80}$, Own_id$_{v\text{-}80}$, Num_vehicles$_{v\text{-}80}$)
Description: Message to leader of platoon indicating vehicle wants to cause a split.

Interface S_send_all:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Sendsplit-ok $_{e\text{-}84}$
Output Action: SEND(idList$_{v\text{-}81}$)
Description: Message from leader to vehicles behind, but ahead of vehicle that initiated the Split maneuver, to proceed with the maneuver.

Interface S_send_splitter:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Sendsplit-ok $_{e\text{-}84}$
Output Action: SEND(Vehicle2_id$_{v\text{-}81}$)
Description: Message from leader to vehicle that initiated the split, indicating that the Split maneuver is in progress.

Interface S_send_complete2:
Source: Vehicle Controller
Destination: Transmitter
Trigger Event: Send_spl_complete2 $_{e\text{-}84}$
Output Action: SEND(Vehicle1_id$_{v\text{-}80}$)
Description: Message from leader to vehicle that initiated the split, indicating completion of the Split maneuver.

A.2 Behavioral Specification

The behavioral state machine part of the specification describes the blackbox behavior of the components, in this case the vehicle controller. Because the specification is blackbox, only externally visible behavior is described and only in terms of external variables. No internal variables or design is included.

VEHICLE CONTROLLER

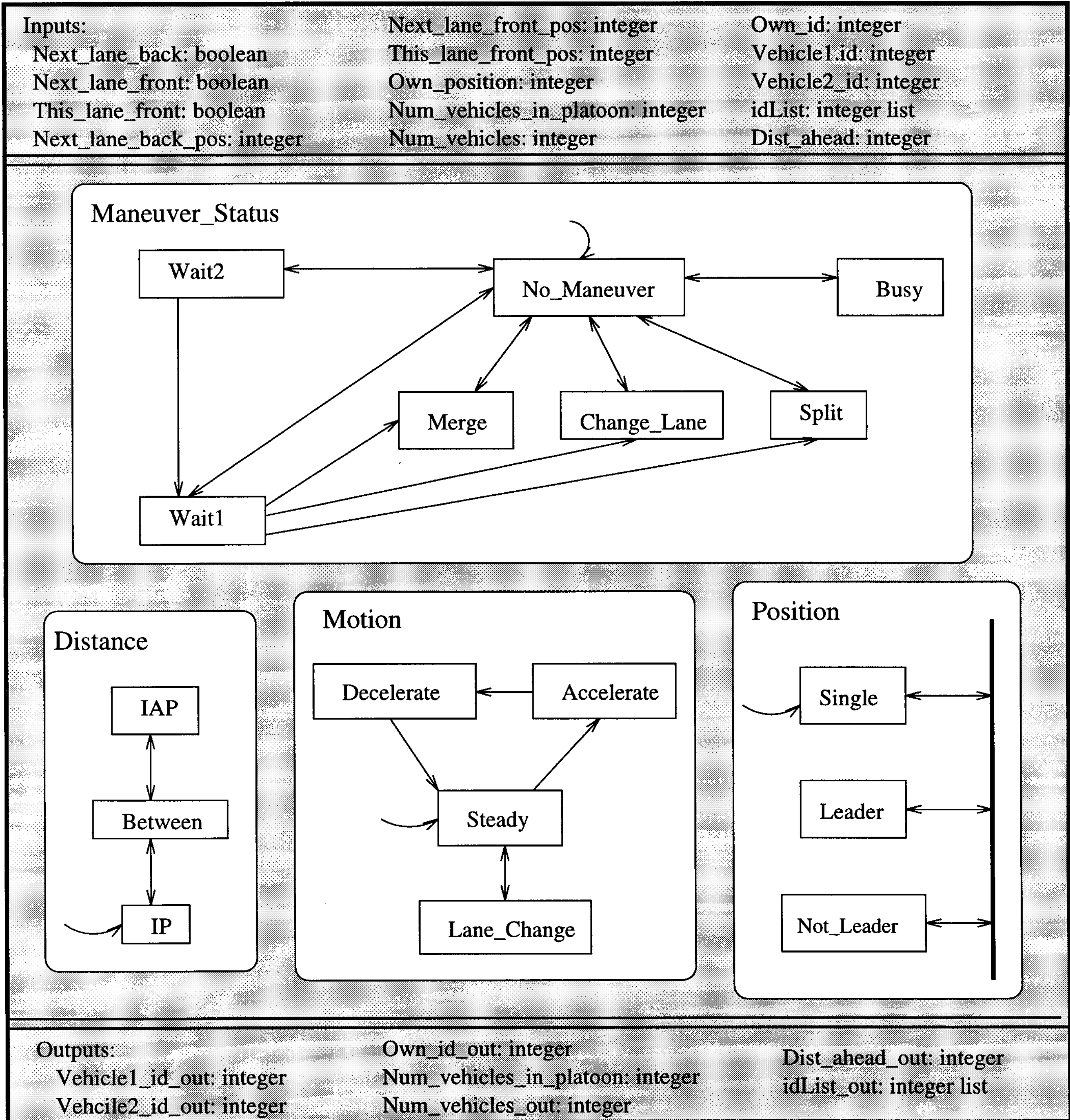


Figure 14: *State-Machine Specification for the Vehicle Controller.*

Maneuver Status Transition Definitions

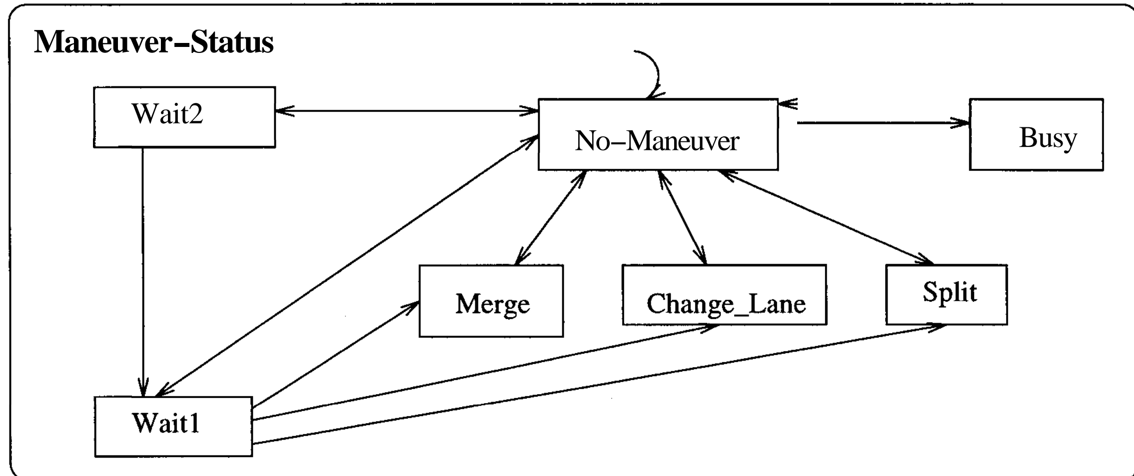


Figure 15:

Transfers to the busy state occur when the vehicle is participating in a maneuver, but it did not initiate that maneuver.

Transition(s): $\boxed{\text{No_Maneuver}_{\text{s-48}}} \longrightarrow \boxed{\text{Busy}_{\text{s-48}}}$

Location: Controller

Trigger Event: $\text{Rcvd_req_chng_lane}_{\text{e-84}}$

Condition:

AND	OR	
$\text{Position}_{\text{s-48}}$ In State $\text{Single}_{\text{s-48}}$	·	T
$\text{Position}_{\text{s-48}}$ In State $\text{Leader}_{\text{s-48}}$	T	·
$\text{Distance}_{\text{s-48}}$ In State $\text{IP}_{\text{s-48}}$	T	T
$\text{Motion}_{\text{s-48}}$ In State $\text{Steady}_{\text{s-48}}$	T	T

Output Action: $\text{Chng_lane_ok}_{\text{e-84}}$, $\text{Set_vehicle2_id}_{\text{e-84}}$

Description: Agree to request from vehicle in adjacent lane to change lanes.

Transition(s): Busy$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$
Location: Controller
Trigger Event: Vehicle_chng_lane_complete$_{e\text{-}84}$
Condition:

AND		OR	
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
	Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T

Output Action:
Description: End of Change-lane maneuver.

Transition(s): Busy$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$
Location: Controller
Trigger Event: TIMEOUT (TIME(PREV(0)Rcvd_req_chng_lane), TIMEOUT_VALUE)$_{e\text{-}85}$
Condition:

AND		OR	
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
	Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T

Output Action:
Description: No response from vehicle changing lanes. Abort Change-lane maneuver.

Transition(s): No_Maneuver$_{s\text{-}48}$ → Busy$_{s\text{-}48}$

Location: Controller

Trigger Event: Rcvd_req_merge$_{e\text{-}84}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T
Num_vehicles$_{v\text{-}80}$ + Num_vehicles_in_platoon$_{v\text{-}80}$ < 20$_{\text{(MAX_VEHICLES_IN_PLATOON)}}$	T	T

Output Action: Merge_ok$_{e\text{-}84}$, Set_vehicle2_id$_{e\text{-}84}$, Set_num_vehicles_p$_{e\text{-}84}$, Set_dist_ahead$_{e\text{-}84}$

Description: Agree to request from platoon behind to merge with this platoon.

Transition(s): Busy$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$

Location: Controller

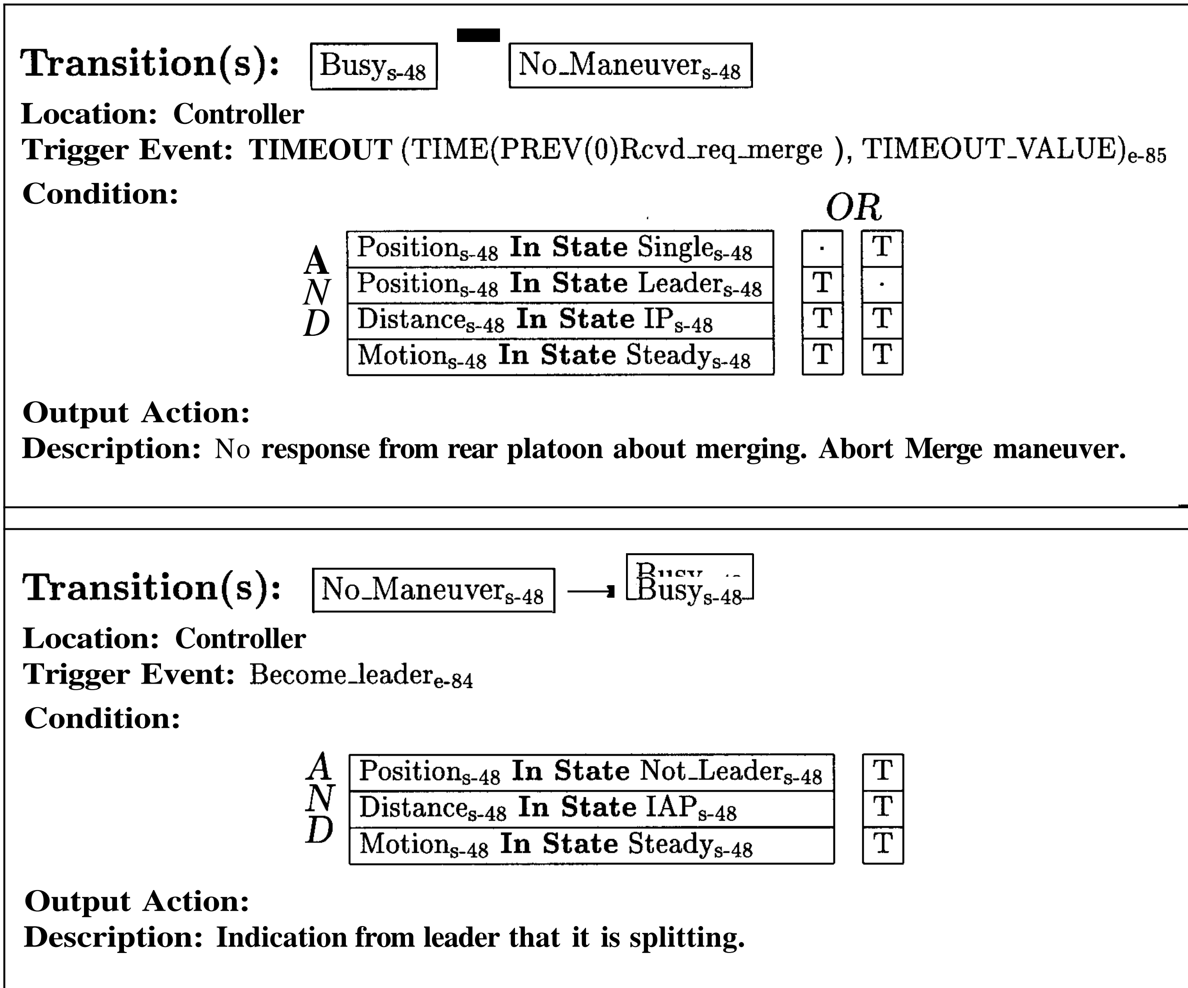
Trigger Event: Vehicle_merge_complete$_{e\text{-}84}$

Condition:

AND	OR	
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·

Output Action:

Description: Rear platoon has completed merge.



Transition(s): $Busy_{s\text{-}48}$ → $No_Maneuver_{s\text{-}48}$

Location: Controller

Trigger Event: **TIMEOUT** (TIME(PREV(0)Rcvd_req_merge), TIMEOUT_VALUE)$_{e\text{-}85}$

Condition:

AND	OR	
$Position_{s\text{-}48}$ **In State** $Single_{s\text{-}48}$	·	T
$Position_{s\text{-}48}$ **In State** $Leader_{s\text{-}48}$	T	·
$Distance_{s\text{-}48}$ **In State** $IP_{s\text{-}48}$	T	T
$Motion_{s\text{-}48}$ **In State** $Steady_{s\text{-}48}$	T	T

Output Action:

Description: No **response from rear platoon about merging. Abort Merge maneuver.**

Transition(s): $No_Maneuver_{s\text{-}48}$ → $Busy_{s\text{-}48}$

Location: Controller

Trigger Event: $Become_leader_{e\text{-}84}$

Condition:

AND	
$Position_{s\text{-}48}$ **In State** $Not_Leader_{s\text{-}48}$	T
$Distance_{s\text{-}48}$ **In State** $IAP_{s\text{-}48}$	T
$Motion_{s\text{-}48}$ **In State** $Steady_{s\text{-}48}$	T

Output Action:

Description: Indication from leader that it is splitting.

Transition(s): Busy$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$

Location: Controller

Trigger Event: Vehicle_split_complete$_{e\text{-}84}$

Condition:

AND	OR	
Distance$_{s\text{-}48}$ **In State** Between$_{s\text{-}48}$	T	T
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T

Output Action:

Description: Former leader of platoon has completed Split maneuver.

Transition(s): No_Maneuver$_{s\text{-}48}$ ⟶ Busy$_{s\text{-}48}$

Location: Controller

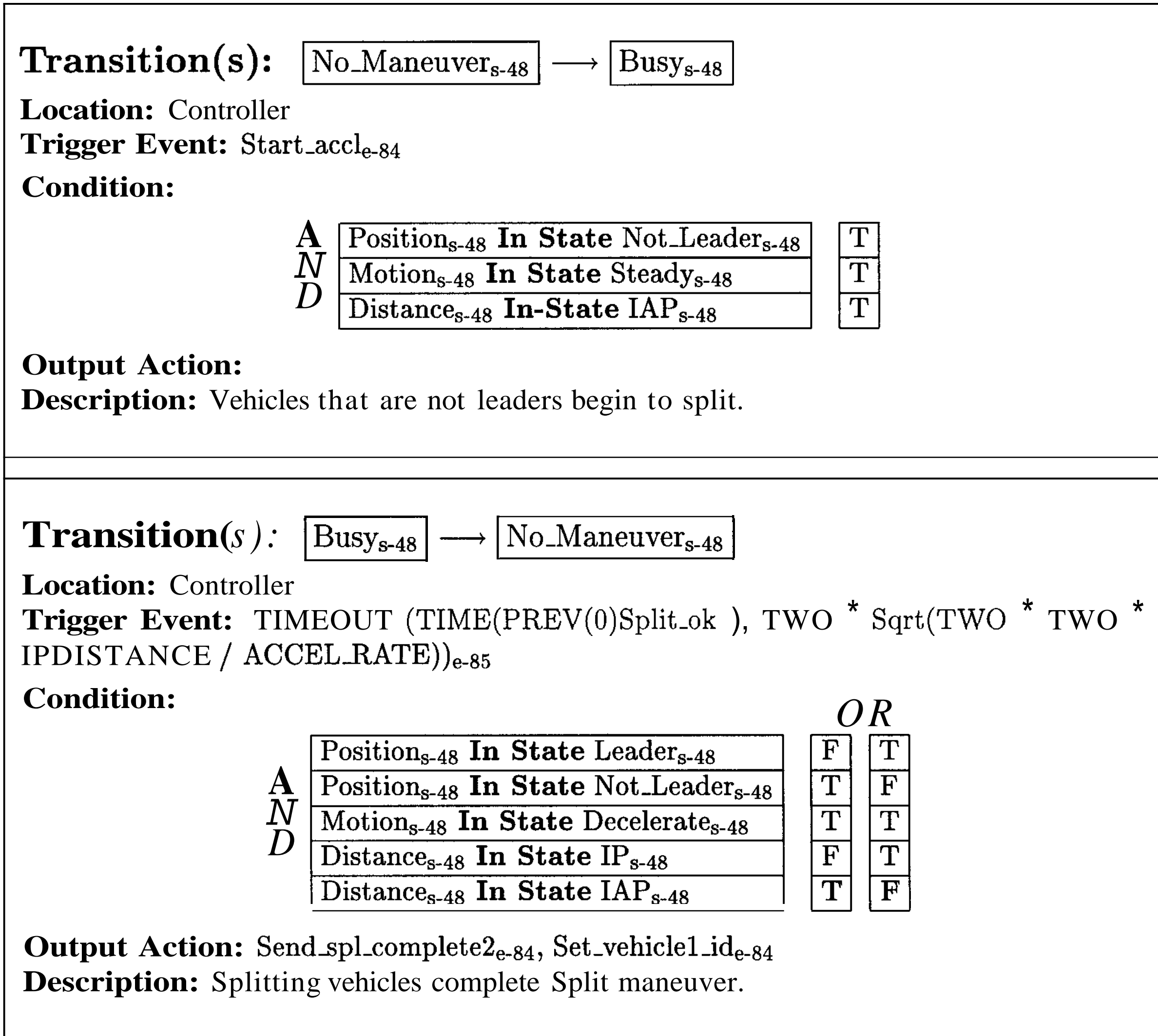
Trigger Event: Rcvd_req_split$_{e\text{-}84}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T
This_lane_front$_{v\text{-}79}$	F	T
This_lane_front_position$_{v\text{-}79}$ − Own_position$_{v\text{-}80}$ $\geq$ 60$_{(\text{MIN_SPLIT_DISTANCE})}$	·	T

Output Action: Split_ok$_{e\text{-}84}$, Send_split_ok$_{e\text{-}84}$, Set_idList$_{e\text{-}84}$, Set_vehicle2_id$_{e\text{-}84}$

Description: Agree to request from vehicle in platoon to cause a split.



Transition(s): No_Maneuver$_{s\text{-}48}$ ⟶ Busy$_{s\text{-}48}$

Location: Controller

Trigger Event: Start_accl$_{e\text{-}84}$

Condition:

AND	
Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
Distance$_{s\text{-}48}$ **In-State** IAP$_{s\text{-}48}$	T

Output Action:

Description: Vehicles that are not leaders begin to split.

Transition(*s*): Busy$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Split_ok), TWO * Sqrt(TWO * TWO * IPDISTANCE / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	F	T
Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T	F
Motion$_{s\text{-}48}$ **In State** Decelerate$_{s\text{-}48}$	T	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	F	T
Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T	F

Output Action: Send_spl_complete2$_{e\text{-}84}$, Set_vehicle1_id$_{e\text{-}84}$

Description: Splitting vehicles complete Split maneuver.

The following transitions represent maneuvers that are initiated by this vehicle.

Transition(s): No_Maneuver$_{s\text{-}48}$ ⟶ Change_Lane$_{s\text{-}48}$

Location: Controller

Trigger Event: System_chng_lane$_{e\text{-}84}$

Condition:

AND	
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
Next_lane_back$_{v\text{-}78}$	F
Next_lane_front$_{v\text{-}79}$	F

Output Action:

Description: Initiate Change-lane maneuver. No vehicles in adjacent lane.

Transition(s): No_Maneuver$_{s\text{-}48}$ ⟶ Wait1$_{s\text{-}48}$

Location: Controller

Trigger Event: System_chng_lane$_{e\text{-}84}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T
Next_lane_back$_{v\text{-}78}$	T	F
Next_lane_front$_{v\text{-}79}$	F	T
Next_lane_front_position$_{v\text{-}79}$ − Own_position$_{v\text{-}80}$ > 20$_{(\text{IP_DISTANCE})}$	·	T
Own_position$_{v\text{-}80}$ − Next_lane_back_position$_{v\text{-}79}$ > 20$_{(\text{IP_DISTANCE})}$	T	·

Output Action: Req_chng_lane1$_{e\text{-}84}$, Set_vehicle1_id$_{e\text{-}84}$, Set_own_id$_{e\text{-}84}$

Description: Send request to vehicle in adjacent lane to change lanes.

Transition(s): Wait1$_{s\text{-}48}$ — Change_Lane$_{s\text{-}48}$

Location: Controller
Trigger Event: Rcvd_chng_lane_ok$_{e\text{-}84}$
Condition:

AND		
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T

Output Action:
Description: Received indication to proceed with changing lanes.

Transition(s): Wait1$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$

Location: Controller
Trigger Event: TIMEOUT (TIME(PREV(0)Req_chng_lane1), TIMEOUT_VALUE)$_{e\text{-}85}$
Condition:

AND		
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T

Output Action:
Description: No response from vehicle in adjacent lane. Abort Change-lane maneuver.

Transition(s): No_Maneuver$_{s\text{-}48}$ $\longrightarrow$ Wait2$_{s\text{-}48}$

Location: Controller

Trigger Event: System_chng_lane$_{e\text{-}84}$

Condition:

AND	
Position$_{s\text{-}48}$ In State Single$_{s\text{-}48}$	T
Distance$_{s\text{-}48}$ In State IP$_{s\text{-}48}$	T
Motion$_{s\text{-}48}$ In State Steady$_{s\text{-}48}$	T
Next_lane_back$_{v\text{-}78}$	T
Next_lane_front$_{v\text{-}79}$	T
Next_lane_front_position$_{v\text{-}79}$ − Own_position$_{v\text{-}80}$ $\geq$ 20$_{(IP_DISTANCE)}$	T
Own_position$_{v\text{-}80}$ − Next_lane_back_position$_{v\text{-}79}$ $\geq$ 20$_{(IP_DISTANCE)}$	T

Output Action: Req_chng_lane2$_{e\text{-}84}$, Set_vehicle1_id$_{e\text{-}84}$, Set_own_id$_{e\text{-}84}$

Description: Send request to two vehicles in adjacent lane to change lanes.

Transition(s): Wait2$_{s\text{-}48}$ — No_Maneuver$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Req_chng_lane2), TIMEOUT_VALUE)$_{e\text{-}85}$

Condition:

AND	
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T

Output Action:

Description: Received indication from both vehicles in adjacent lane to proceed with changing lanes.

Transition(s): Wait2$_{s\text{-}48}$ ▬ Wait1$_{s\text{-}48}$

Location: Controller

Trigger Event: Rcvd_chng_lane_ok$_{e\text{-}84}$

Condition:

AND	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T

Output Action:

Description: Received indication from one vehicle to proceed with changing lanes. Wait for reply from the other.

Transition(s): Change_Lane$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)System_chng_lane), CHANGE_LANE_TIME)$_{e\text{-}84}$

Condition:

AND	Motion$_{s\text{-}48}$ **In State** Lane_Change$_{s\text{-}48}$	T
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T

Output Action: Change_lane_complete$_{e\text{-}84}$, Set_vehicle1_id$_{e\text{-}84}$, Set_vehicle2_id$_{e\text{-}84}$

Description: Vehicle changing lanes completes Change-lane maneuver.

Transition(s): No_Maneuver$_{s\text{-}48}$ $\longrightarrow$ Merge$_{s\text{-}48}$

Location: Controller

Trigger Event: Rcvd_merge_ok$_{e\text{-}84}$

Condition:

AND	
Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T

Output Action:

Description: Platoon ahead has agreed to Merge maneuver. Begin merging.

Transition(s) : No_Maneuver$_{s\text{-}48}$ $\longrightarrow$ Wait1$_{s\text{-}48}$

Location: Controller

Trigger Event: System_merge$_{e\text{-}84}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T
Dist_ahead$_{v\text{-}81}$ $\geq$ 20$_{(\text{IP_DISTANCE})}$	T	T

Output Action: Req_merge$_{e\text{-}84}$, Set_vehicle1_id$_{e\text{-}84}$, Set_own_id$_{e\text{-}84}$, Set_num_vehicles_p$_{e\text{-}84}$, Set_dist_ahead$_{e\text{-}84}$

Description: Send request to platoon ahead to merge with it.

Transition(s): Wait1$_{s\text{-}48}$ ⟶ Merge$_{s\text{-}48}$

Location: Controller

Trigger Event: Rcvd_merge_ok$_{e\text{-}84}$

Condition:

AND		OR	
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
	Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T

Output Action:

Description: Received indication to proceed with merging with platoon ahead.

Transition(s): Wait1$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Req_merge), TIMEOUT_VALUE)$_{e\text{-}85}$

Condition:

AND		OR	
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
	Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T

Output Action:

Description: No response from platoon ahead. Abort Merge maneuver.

Transition(s): Merge$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Rcvd_merge_ok), TWO * Sqrt(TWO * Dist-ahead / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND		OR		
Position$_{s\text{-}48}$ In State Single$_{s\text{-}48}$	·	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	·	T	·
Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T	·	·
Distance$_{s\text{-}48}$ In State Between$_{s\text{-}48}$	·	T	T
Distance$_{s\text{-}48}$ In State IAP$_{s\text{-}48}$	T	·	·
Motion$_{s\text{-}48}$ **In State** Decelerate$_{s\text{-}48}$	T	T	T

Output Action:

Description: Platoon merges with platoon ahead.

Transition(s) : No_Maneuver$_{s\text{-}48}$ ⟶ Split$_{s\text{-}48}$

Location: Controller

Trigger Event: System_split$_{e\text{-}84}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T
This_lane_front$_{v\text{-}79}$	F	T
This_lane_front_position$_{v\text{-}79}$ $-$ Own_position$_{v\text{-}80}$ $\geq 60_{(MIN_SPLIT_DISTANCE)}$	·	T

Output Action: Start_leader_split$_{e\text{-}84}$, Send_leader_split$_{e\text{-}84}$, Set_vehicle1_id$_{e\text{-}84}$

Description: Leader initiates Split maneuver.

Transition(s): No_Maneuver$_{s\text{-}48}$ ⟶ Wait1$_{s\text{-}48}$

Location: Controller

Trigger Event: System_split$_{e\text{-}84}$

Condition:

AND		
	Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T

Output Action: Req_split$_{e\text{-}84}$, Set_vehicle1_id$_{e\text{-}84}$, Set_own_id$_{e\text{-}84}$, Set_num_vehicles$_{e\text{-}84}$

Description: Send request to leader to split platoon.

Transition(s): Wait1$_{s\text{-}48}$ ⟶ Split$_{s\text{-}48}$

Location: Controller

Trigger Event: Rcvd_split_ok$_{e\text{-}84}$

Condition:

AND		
	Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T

Output Action:

Description: Vehicles ahead start to split from rest of platoon.

Transition(s): Split$_{s\text{-}48}$ ⟶ No_Maneuver$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Start_leader_split), TWO * Sqrt(TWO * IPDISTANCE / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND		
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Decelerate$_{s\text{-}48}$	T

Output Action:

Description: Complete Split maneuver.

Transition(s): Split$_{s\text{-}48}$ → No_Maneuver$_{s\text{-}48}$

Location: Controller

Trigger Event: Vehicle_split_complete$_{e\text{-}84}$

Condition:

AND	OR	
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T
Distance$_{s\text{-}48}$ **In State** Between$_{s\text{-}48}$	T	T
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·

Output Action:

Description: Complete Split maneuver.

Motion Transition Definitions

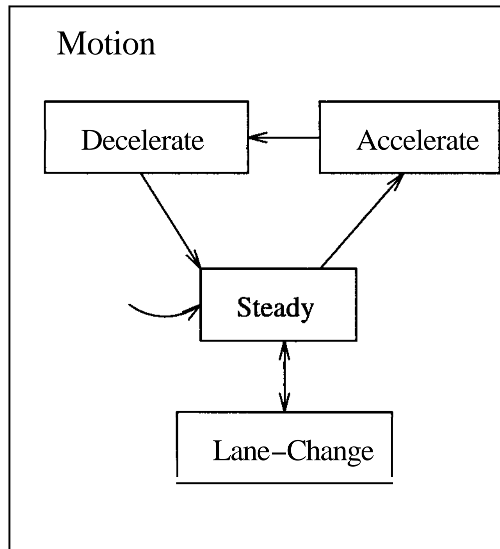


Figure 16:

Transition(s): Accelerate$_{s\text{-}48}$ ⟶ Decelerate$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Rcvd_merge_ok), TWO * Sqrt(TWO * Dist_ahead / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND	OR		
Position$_{s\text{-}48}$ In State Single$_{s\text{-}48}$	·	·	T
Position$_{s\text{-}48}$ In State Leader$_{s\text{-}48}$	·	T	·
Position$_{s\text{-}48}$ In State Not_Leader$_{s\text{-}48}$	T	·	·
Maneuver_Status$_{s\text{-}48}$ In State Merge$_{s\text{-}48}$	T	T	T

Output Action:

Description: Decelerate during Merge maneuver.

Transition(s): Accelerate$_{s\text{-}48}$ ⟶ Decelerate$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Start_leader_split), Sqrt(TWO * IPDISTANCE / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND	
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T
Maneuver_Status$_{s\text{-}48}$ **In State** Split$_{s\text{-}48}$	T

Output Action:

Description: Former leader of platoon decelerates during Split maneuver.

Transition(s): Accelerate$_{s\text{-}48}$ ⟶ Decelerate$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Split_ok), Sqrt(TWO * TWO * IPDISTANCE / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	F	T
Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T	F
Maneuver_Status$_{s\text{-}48}$ **In State** Busy$_{s\text{-}48}$	T	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	F	T
Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T	F

Output Action:

Description: Vehicles decelerate during Split maneuver.

Transition(s): Decelerate$_{s\text{-}48}$ ⟶ Steady$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Rcvd_merge_ok), TWO * Sqrt(TWO * Dist_ahead / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND	OR		
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	·	T	·
Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T	·	·
Maneuver_Status$_{s\text{-}48}$ **In State** Merge$_{s\text{-}48}$	T	T	T
Distance$_{s\text{-}48}$ **In State** Between$_{s\text{-}48}$	·	·	T
Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	·	T	·
Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T	·	·

Output Action: Send_merge_complete$_{e\text{-}84}$, Set_vehicle1_id$_{e\text{-}84}$

Description: Vehicles have merged with platoon ahead.

Transition(s): Decelerate$_{s-48}$ — Steady$_{s-48}$

Location: Controller
Trigger Event: TIMEOUT (TIME(PREV(0)Start_leader_split), TWO * Sqrt(TWO * IPDISTANCE / ACCEL_RATE))$_{e-85}$
Condition:

AND	
Position$_{s-48}$ **In State** Single$_{s-48}$	T
Distance$_{s-48}$ **In State** IP$_{s-48}$	T
Maneuver_Status$_{s-48}$ **In State** Split$_{s-48}$	T

Output Action: Send_spl_complete1$_{e-84}$, Set_vehicle1_id$_{e-84}$
Description: Former leader has completed Split maneuver.

Transition(s): Decelerate$_{s-48}$ ⟶ Steady$_{s-48}$

Location: Controller
Trigger Event: TIMEOUT (TIME(PREV(0)Split_ok), TWO * Sqrt(TWO * TWO * IPDISTANCE / ACCEL_RATE))$_{e-85}$
Condition:

AND	OR	
Position$_{s-48}$ **In State** Leader$_{s-48}$	F	T
Position$_{s-48}$ **In State**Not_Leader$_{s-48}$	T	F
Maneuver_Status$_{s-48}$ **In State** Busy$_{s-48}$	T	T
Distance$_{s-48}$ **In State** IP$_{s-48}$	F	T
Distance$_{s-48}$ **In State** IAP$_{s-48}$	T	F

Output Action:
Description: Vehicles have completed Split maneuver.

Transition(s): Lane_Change$_{s\text{-}48}$ $\longrightarrow$ Steady$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)System_chng_lane), CHANGE_LANE_TIME)$_{e\text{-}84}$

Condition:

AND		
	Maneuver_Status$_{s\text{-}48}$ In State Change_Lane$_{s\text{-}48}$	T
	Position$_{s\text{-}48}$ In State Single$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ In State IP$_{s\text{-}48}$	T

Output Action:

Description: Completion of Change-lane maneuver.

Transition(s): Steady$_{s\text{-}48}$ $\longrightarrow$ Accelerate$_{s\text{-}48}$

Location: Controller

Trigger Event: Rcvd_merge_ok$_{e\text{-}84}$

Condition:

AND		OR		
	Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	·	·	T
	Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T	·
	Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T	·	·
	Maneuver_Status$_{s\text{-}48}$ **In State** Wait1$_{s\text{-}48}$	·	T	T
	Maneuver_Status$_{s\text{-}48}$ **In State** No_Maneuver$_{s\text{-}48}$	T	·	·
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	F	T	T
	Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T	F	F

Output Action:

Description: Start accelerating towards platoon ahead to merge with it.

Transition(s): Steady$_{s\text{-}48}$ ⟶ Accelerate$_{s\text{-}48}$

Location: Controller

Trigger Event: Start_leader_split$_{e\text{-}84}$

Condition:

	Condition	
A	Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T
N	Maneuver_Status$_{s\text{-}48}$ **In State** Split$_{s\text{-}48}$	T
D	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T

Output Action:

Description: Leader starts accelerating away from platoon during Split maneuver.

Transition(s): Steady$_{s\text{-}48}$ ⟶ Accelerate$_{s\text{-}48}$

Location: Controller

Trigger Event: Split_ok$_{e\text{-}84}$

Condition:

	Condition	
A	Maneuver_Status$_{s\text{-}48}$ **In State** Busy$_{s\text{-}48}$	T
N	Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T
D	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T

Output Action:

Description: Leader starts accelerating during Split maneuver.

Transition(s): Steady$_{s\text{-}48}$ Accelerate$_{s\text{-}48}$

Location: Controller

Trigger Event: Start_accl$_{e\text{-}84}$

Condition:

	Condition	
A	Position$_{s\text{-}48}$ **In State** Not_Leader$_{s\text{-}48}$	T
N	Maneuver_Status$_{s\text{-}48}$ **In State** No_Maneuver$_{s\text{-}48}$	T
D	Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T

Output Action:

Description: Vehicles that are not leaders start accelerating during Split maneuver.

Transition(s): Steady$_{\text{s-48}}$ ⟶ Lane_Change$_{\text{s-48}}$

Location: Controller

Trigger Event: System_chng_lane$_{\text{e-84}}$

Condition:

AND		
	Maneuver_Status$_{\text{s-48}}$ **In State** No_Maneuver$_{\text{s-48}}$	T
	Position$_{\text{s-48}}$ **In State** Single$_{\text{s-48}}$	T
	Distance$_{\text{s-48}}$ **In State** IP$_{\text{s-48}}$	T
	Next_lane_back$_{\text{v-78}}$	F
	Next_lane_front$_{\text{v-79}}$	F

Output Action:

Description: No vehicles in adjacent lane. Proceed with changing lanes.

Transition(s): Steady$_{\text{s-48}}$ ⟶ Lane_Change$_{\text{s-48}}$

Location: Controller

Trigger Event: Rcvd_chng_lane_ok$_{\text{e-84}}$

Condition:

AND		
	Maneuver_Status$_{\text{s-48}}$ **In State** Wait1$_{\text{s-48}}$	T
	Position$_{\text{s-48}}$ **In State** Single$_{\text{s-48}}$	T
	Distance$_{\text{s-48}}$ **In State** IP$_{\text{s-48}}$	T

Output Action:

Description: Received indication to proceed with changing lanes.

Distance Transition Definitions

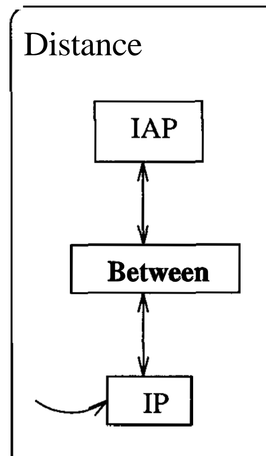


Figure 17:

Transition(s): $\boxed{\text{Between}_{s\text{-}48}} \longrightarrow \boxed{\text{IAP}_{s\text{-}48}}$
Location: Controller
Trigger Event: $\text{Received_lane_info}_{e\text{-}84}$
Condition:

Output Action:
Description: Act on sensor input.

Transition(s): Between$_{s\text{-}48}$ ⟶ IAP$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Rcvd_merge_ok), TWO * Sqrt(TWO * Dist-ahead / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
Maneuver_Status$_{s\text{-}48}$ **In State** Merge$_{s\text{-}48}$	T	T
Motion$_{s\text{-}48}$ **In State** Decelerate$_{s\text{-}48}$	T	T

Output Action:

Description: Towards completion of Merge maneuver.

Transition(s): Between$_{s\text{-}48}$ ⟶ IP$_{s\text{-}48}$

Location: Controller

Trigger Event: Vehicle_split_complete$_{e\text{-}84}$

Condition:

AND	OR	
Maneuver_Status$_{s\text{-}48}$ **In State** Busy$_{s\text{-}48}$	T	T
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T

Output Action:

Description: Former leader of platoon has accelerated away to complete split.

Transition(s): Between$_{s-48}$ ⟶ IP$_{s-48}$

Location: Controller

Trigger Event: Vehicle_split_complete$_{e-84}$

Condition:

AND	OR	
Motion$_{s-48}$ **In State** Steady$_{s-48}$	T	T
Maneuver_Status$_{s-48}$ **In State** Split$_{s-48}$	T	T
Position$_{s-48}$ **In State** Single$_{s-48}$	·	T
Position$_{s-48}$ **In State** Leader$_{s-48}$	T	·

Output Action:

Description: Vehicles ahead have accelerated away to complete split.

Transition(s): IAP$_{s-48}$ ▬ Between$_{s-48}$

Location: Controller

Trigger Event: Become_leader$_{e-84}$

Condition:

AND	
Position$_{s-48}$ **In State** Not_Leader$_{s-48}$	T
Maneuver_Status$_{s-48}$ **In State** No_Maneuver$_{s-48}$	T
Motion$_{s-48}$ **In State** Steady$_{s-48}$	T

Output Action:

Description: Vehicle in front, i.e. the leader, starts splitting away.

Transition(s): IAP$_{s-48}$ ▬ Between$_{s-48}$

Location: Controller

Trigger Event: Rcvd_split_ok$_{e-84}$

Condition:

AND	
Maneuver_Status$_{s-48}$ **In State** Wait1$_{s-48}$	T
Position$_{s-48}$ **In State** Not_Leader$_{s-48}$	T
Motion$_{s-48}$ **In State** Steady$_{s-48}$	T

Output Action:

Description: Vehicles ahead start splitting away.

Transition(s): $\boxed{IP_{s\text{-}48}} \longrightarrow \boxed{Between_{s\text{-}48}}$

Location: Controller

Trigger Event: Rcvd_merge_ok$_{e\text{-}84}$

Condition:

AND	OR	
Position$_{s\text{-}48}$ **In State** Single$_{s\text{-}48}$	·	T
Position$_{s\text{-}48}$ **In State** Leader$_{s\text{-}48}$	T	·
Maneuver_Status$_{s\text{-}48}$ **In State** Wait1$_{s\text{-}48}$	T	T
Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T	T
Dist_ahead$_{v\text{-}81}$ = 20$_{(IP_DISTANCE)}$	T	T

Output Action:

Description: Start merging with platoon ahead.

Position Transition Definitions

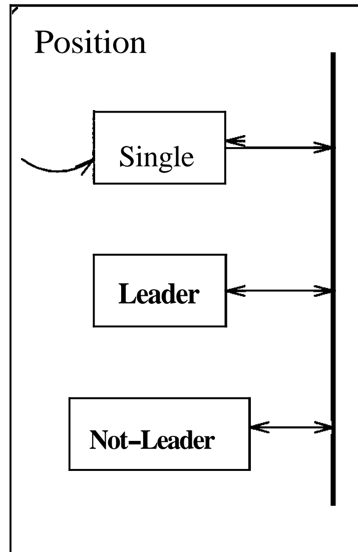


Figure 18:

Transition(s): Leader$_{s\text{-}48}$ ⟶ Not_Leader$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Rcvd_merge_ok), TWO * Sqrt(TWO * Dist_ahead / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND	
Maneuver_Status$_{s\text{-}48}$ **In State** Merge$_{s\text{-}48}$	T
Motion$_{s\text{-}48}$ **In State** Decelerate$_{s\text{-}48}$	T
Distance$_{s\text{-}48}$ **In State**Between$_{s\text{-}48}$	T

Output Action:

Description: Platoon has merged with platoon ahead.

Transition(s): $\boxed{\text{Leader}_{s\text{-}48}} \longrightarrow \boxed{\text{Single}_{s\text{-}48}}$

Location: Controller

Trigger Event: $\text{Start_leader_split}_{e\text{-}84}$

Condition:

AND		
A	$\text{Motion}_{s\text{-}48}$ **In State** $\text{Steady}_{s\text{-}48}$	T
N	$\text{Maneuver_Status}_{s\text{-}48}$ **In State** $\text{Split}_{s\text{-}48}$	T
D	$\text{Distance}_{s\text{-}48}$ **In State** $\text{IP}_{s\text{-}48}$	T

Output Action:

Description: Leader begins to split away from platoon.

Transition(s): $\boxed{\text{Leader}_{s\text{-}48}} \longrightarrow \boxed{\text{Single}_{s\text{-}48}}$

Location: Controller

Trigger Event: $\text{Split_ok}_{e\text{-}84}$

Condition:

AND		
A	$\text{Distance}_{s\text{-}48}$ **In State** $\text{IP}_{s\text{-}48}$	T
N	$\text{Maneuver_Status}_{s\text{-}48}$ **In State** $\text{Busy}_{s\text{-}48}$	T
D	$\text{Motion}_{s\text{-}48}$ **In State** $\text{Steady}_{s\text{-}48}$	T
	$\text{Num_vehicles_in_platoon}_{v\text{-}80} - \text{Num_vehicles}_{v\text{-}80} = 1$	T

Output Action:

Description: Leader begins to split. This is the only other vehicle in the platoon.

Transition(s): Not_Leader$_{s\text{-}48}$ ⟶ Leader$_{s\text{-}48}$

Location: Controller

Trigger Event: Become_leader$_{e\text{-}84}$

Condition:

	Condition	
AND	Maneuver_Status$_{s\text{-}48}$ **In State** No_Maneuver$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
	Num_vehicles_in_platoon$_{v\text{-}80}$ > 2	T

Output Action:

Description: Become new leader of platoon as leader splits.

Transition(s): Not_Leader$_{s\text{-}48}$ ⟶ Leader$_{s\text{-}48}$

Location: Controller

Trigger Event: Rcvd_split_ok$_{e\text{-}84}$

Condition:

	Condition	
AND	Maneuver_Status$_{s\text{-}48}$ **In State** Wait1$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
	Num_vehicles$_{v\text{-}80}$ > 0	T

Output Action:

Description: Become new leader of platoon as part of it accelerates away during Split maneuver.

Transition(s): Not_Leader$_{s\text{-}48}$ ⟶ Single$_{s\text{-}48}$

Location: Controller

Trigger Event: Become_leader$_{e\text{-}84}$

Condition:

AND		
	Maneuver_Status$_{s\text{-}48}$ **In State** No_Maneuver$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State**IAP$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State**Steady$_{s\text{-}48}$	T
	Num_vehicles_in_platoon$_{v\text{-}80}$ = 2	T

Output Action:

Description: Leader splits and this is the only other vehicle in the platoon.

Transition(s): Not_Leader$_{s\text{-}48}$ ⟶ Single$_{s\text{-}48}$

Location: Controller

Trigger Event: Rcvd_split_ok$_{e\text{-}84}$

Condition:

AND		
	Maneuver_Status$_{s\text{-}48}$ **In State** Wait1$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IAP$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
	Num_vehicles$_{v\text{-}80}$ = 0	T

Output Action:

Description: Rest of platoon splits and this is the only vehicle left.

Transition(s): Single$_{s\text{-}48}$ ⟶ Leader$_{s\text{-}48}$

Location: Controller

Trigger Event: Vehicle_merge_complete$_{e\text{-}84}$

Condition:

AND		
	Maneuver_Status$_{s\text{-}48}$ **In State** Busy$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Steady$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** IP$_{s\text{-}48}$	T

Output Action:

Description: Rear platoon merges with this vehicle.

Transition(s): Single$_{s\text{-}48}$ ⟶ Not_Leader$_{s\text{-}48}$

Location: Controller

Trigger Event: TIMEOUT (TIME(PREV(0)Rcvd_merge_ok), TWO * Sqrt(TWO * Dist_ahead / ACCEL_RATE))$_{e\text{-}85}$

Condition:

AND		
	Maneuver_Status$_{s\text{-}48}$ **In State** Merge$_{s\text{-}48}$	T
	Motion$_{s\text{-}48}$ **In State** Decelerate$_{s\text{-}48}$	T
	Distance$_{s\text{-}48}$ **In State** Between$_{s\text{-}48}$	T

Output Action:

Description: Merge with platoon ahead.

A.3 Input Variables

Input: Next-lane-back

Type: boolean
Expected Range: *True, False*
Granularity: N/A
Units: N/A
Load:
Exception handling information:
Description: *True* if vehicle present behind in adjacent lane, *False* otherwise.

Input: Next_lanefront
Type: boolean
Expected Range: *True, False*
Granularity: N/A
Units: N/A
Load:
Exception handling information:
Description: *True* if vehicle present ahead in adjacent lane, *False* otherwise.

Input: Thislane-front
Type: boolean
Expected Range: *True, False*
Granularity: N/A
Units: N/A
Load:
Exception handling information:
Description: *True* if vehicle present ahead in own lane, *False* otherwise.

Input: Next-lane-back-position
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: Position of vehicle behind in adjacent lane.

Input: Next-lanefront-position
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: Position of vehicle ahead in adjacent lane.

Input: This-lane-front-position
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: Position of vehicle ahead in own lane.

Input: Own-position
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: Position of own vehicle.

Input: Num_vehicles_in_platoon
Type: integer
Expected Range: 0 .. 20
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: Number of vehicles in platoon.

Input: Num_vehicles
Type: integer
Expected Range: 0 .. 20
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: Number of vehicles in a platoon. Used in communication during Merge or Split maneuver.

Input: Own_id
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: ID of own vehicle.

Input: Vehicleid
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: ID of a vehicle.

Input: Vehicle2_id
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: ID of a vehicle.

Input: idList
Type: integer list
Expected Range: 0 .. 1000
Granularity: 1
Units: N/A
Load:
Exception handling information:
Description: List of IDs of vehicles in platoon.

Input: Dist_ahead
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Units: feet
Load:
Exception handling information:
Description: Distance between vehicle and one ahead. Used in Merge.

A.4 Output variables

Output: Vehiclelid-out
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Trigger: $\text{Set_vehicle1_id}_{\text{e-84}}$
Assignment: $\text{Vehicle1_id}_{\text{v-80}}$
Units: N/A
Load:
Exception handling information:
Description: ID of vehicle.

Output: Vehicle2_id_out
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Trigger: $\text{Set_vehicle2_id}_{\text{e-84}}$
Assignment: $\text{Vehicle2_id}_{\text{v-81}}$
Units: N/A
Load:
Exception handling information:
Description: ID of vehicle.

Output: Own-id-out
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Trigger: $\text{Set_own_id}_{\text{e-84}}$
Assignment: $\text{Own_id}_{\text{v-80}}$
Units: N/A
Load:
Exception handling information:
Description: ID of own vehicle.

Output: Num_vehicles_in_platoon_out
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Trigger: $\text{Set_num_vehicles_p}_{\text{e-84}}$
Assignment: $\text{Num_vehicles_in_platoon}_{\text{v-80}}$
Units: N/A
Load:
Exception handling information:
Description: Number of vehicles in own platoon.

Output: Num_vehicles_out
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Trigger: $\text{Set_num_vehicles}_{\text{e-84}}$
Assignment: $\text{Num_vehicles}_{\text{v-80}} + 1$
Units: N/A
Load:
Exception handling information:
Description: Number of vehicles in platoon, plus one.

Output: Dist_ahead_out
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Trigger: $\text{Set_dist_ahead}_{\text{e-84}}$
Assignment: $\text{Dist_ahead}_{\text{v-81}}$
Units: feet
Load:
Exception handling information:
Description: Distance of vehicle ahead.

Output: idList_out
Type: integer
Expected Range: 0 .. 1000
Granularity: 1
Trigger: $\text{Set_idList}_{\text{e-84}}$
Assignment: $\text{idList}_{\text{v-81}}$
Units: N/A
Load:
Exception handling information:
Description: List of ids of vehicles in platoon.

A.5 Constants

IP-DISTANCE : 20
CHANGE-LANE-TIME : 3
TIMEOUT-VALUE : 10
IAP_DISTANCE : 2
ACCEL-RATE : 5
MAX-VEHICLES-IN_PLATOON : 20
MIN_SPLIT_DISTANCE : 60

A.6 Event List

System_chng_lane
Rcvd_req_chng_lane
Vehicle-chnglane-complete
Rcvd_chng_lane_ok
Change_lane_complete
Req_chng_lane1
Chng_lane_ok
Req_chng_lane2
Systemmerge
Rcvd_req_merge
Rcvdmerge-ok
Vehiclemerge-complete
Sendmerge-complete
Reqmerge
Merge-ok
Systemsplit
Start_leader_split
Become_leader
Rcvd_req_split
Start_accl
Rcvd_split_ok
Split_ok
Vehiclesplit-complete
Split2_complete
Send_leader_split
Reqsplit
Sendsplit-ok
Sendspl-complete1
Send_spl_complete2
Received_lane_info
Set_vehicle1_id
Set_vehicle2_id
Set_own_id
Setaurn-vehicles
Set_num_vehicles_p
Set_dist_ahead
Set_idList
TIMEOUT (TIME(PREV(0)System_chng_lane), CHANGE-LANE-TIME)
TIMEOUT (TIME(PREV(0)Rcvd_req_chng_lane), TIMEOUT-VALUE)

TIMEOUT (TIME(PREV(0)Req_chng_lane1), TIMEOUT-VALUE)
TIMEOUT (TIME(PREV(0)Req_chng_lane2), TIMEOUT-VALUE)
TIMEOUT (TIME(PREV(0)Req_merge), TIMEOUT-VALUE)
TIMEOUT (TIME(PREV(0)Rcvd_merge_ok), TWO * Sqrt(TWO * Dist_ahead
/ ACCEL-RATE))
TIMEOUT (TIME(PREV(0)Rcvd_req_merge), TIMEOUT-VALUE)
TIMEOUT (TIME(PREV(0)Start_leader_split), Sqrt(TWO * IP-DISTANCE
/ ACCEL-RATE))
TIMEOUT (TIME(PREV(0)Start_leader_split), TWO * Sqrt(TWO * IPDISTANCE
/ ACCEL-RATE))
TIMEOUT (TIME(PREV(0)Split_ok), Sqrt(TWO * TWO * IP-DISTANCE
/ ACCEL-RATE))
TIMEOUT (TIME(PREV(0)Split_ok), TWO * Sqrt(TWO * TWO * IP-DISTANCE
/ ACCEL-RATE))

UNIVERSITY OF CALIFORNIA

IRVINE

Software Deviation Analysis

A dissertation submitted in partial satisfaction of the

requirements for the degree of Doctor of Philosophy

in Information and Computer Science

by

Jon Damon Reese

Committee in charge:
Professor John King, Chair
Professor Nancy G. Leveson
Professor Debra J. Richardson

1996

© Jon Damon Reese, 1996.
All rights reserved.

The dissertation of Jon Damon Reese is approved,

and is acceptable in quality and form for

publication on microfilm:

____

____

____

Committee Chair

University of California, Irvine

1996

Dedication

This dissertation is dedicated to my grandparents—
James Dunn, Doris Wilson Dunn (may she rest in peace),
Edison Reese (may he rest in peace), and
Leone Marshall Reese.

Contents

List of Figures vi

List of Tables viii

Acknowledgement ix

Curriculum Vitae x

Abstract xi

Chapter 1 Introduction 1

1.1 Contribution of the dissertation 5

1.2 Organization of the dissertation 6

Chapter 2 Related Work 7

2.1 Backward Search Techniques 11

2.2 Forward Search Techniques 21

2.3 Combined Techniques 26

2.4 Summary 32

Chapter 3 Deviation Analysis 33

3.1 Goals 40

Chapter 4 A Primitive Language of Causality 41

4.1 Definitions 41

4.2 Encoding Causality 49

4.3 Causality Diagram Grammar 51

4.4 Translation of RSML 55

4.5 Summary 67

Chapter 5 A Calculus of Deviations 68

5.1 Introduction To Qualitative Mathematics 69

5.2 $P_{B,N}$: A Logarithmic Qualitative Domain 72

5.3 A Qualitative Calculus for $P_{B,N}$ 77

5.4 Inverse Relations 92

5.5 A Qualitative Calculus of Deviations 93

5.6 Deviation Formulae 95

5.7 Application of Deviation Formulae to $P_{B,N}$ 102

5.8 Assumptive Functions 103

5.9 States 105

Chapter 6 A Forward Search Algorithm **109**
6.1 Semi-Automated Analysis 111
6.2 Fully-Automated Analysis 113
6.3 Propagation of Definite Deviations 116
6.4 Propagation of Possible Deviations 128
6.5 Summary 131

Chapter 7 Examples **133**
7.1 Train Crossing 135
7.2 TCAS II 140
7.3 Conclusions 147

Chapter 8 Results and Future Directions **155**
8.1 Results 155
8.2 Future Directions 157

Bibliography **160**

List of Figures

2.1 The TCAS II model of detected aircraft. written in RSML 9
2.2 Fault tree 13
2.3 Mini-fault tree for IF-THEN-ELSE 15
2.4 Mini-fault tree constructed from standard component transition table 16
2.5 An instantiated mini-fault tree 17
2.6 A simple digraph 18
2.7 Conversion of pipe-and-process diagram to digraph 19
2.8 Example of a FMECA table 22
2.9 An event tree 24
2.10 An event tree for two unordered events 25
2.11 A cause-consequence diagram 27

3.1 Receipt of Mode S Address from the Mode S transponder 34
3.2 TCAS input interface 36
3.3 Macro *RA-Display-Delay* 37
3.4 Transition to *TA/RA-Delay* upon first entering state *Threat* 37
3.5 Transition to ***RA*** upon first entering state *Threat* 38
3.6 Transition from *TA/RA-Delay* to ***RA*** 39

4.1 Causality diagram example 54
4.2 The *Intruder-Status* state heirarchy 57
4.3 A causality diagram fragment for *Threat* 58
4.4 A causality diagram fragment for *Proximate-Traffic* 59
4.5 Example of a causality digram fragment for an RSML transition . . . 61
4.6 Explicit representation of default transitions 62
4.7 AND/OR Table 63
4.8 Causality diagram template for input variables 65
4.9 Causality diagram template for output variables 65
4.10 Causality diagram template for input interfaces 66

5.1 A qualitative representation of oscillating functions 71
5.2 Example of how assumptions and sequential propagation relate 107
5.3 Algorithms for determining whether a mode contains any deviations. 108

6.1 A semi-automated search procedure 110
6.2 A fully-automated search procedure 112
6.3 An example of a search tree produced by the `Automated` procedure . . 115

6.4 The Propagate_Definite_Deviations procedure. 117
6.5 The Propagate_Forward_Definite procedure. 122
6.6 The PropagateBackwardDefinite procedure. 124
6.7 Sample C++ functions for calculating equivalence code. 127
6.8 The PropagateBackwardDefinite procedure. 129

7.1 Train crossing example. 134
7.2 Queue sizes on train crossing example 1. 135
7.3 Assumption depth of Automated on train crossing example 137
7.4 Step depth of Automated on train crossing example 138
7.5 Search queue comparison for train crossing examples 1 and 2. 139
7.6 Assumption comparison for train crossing examples 1 and 2. 139
7.7 Search queue comparison for train crossing examples 1 and 3. 140
7.8 The TCAS II model of its own aircraft. 141
7.9 Queue sizes on TCAS II example. 142
7.10 The automatic sensitivity level transition from state 1 to state 5. . . . 144
7.11 Search order of Automated on the TCAS II example. 151

List of Tables

2.1 HAZOP guide words 29
2.2 MASCOT guide words 30
2.3 Summary of hazard analysis methods 32

4.1 Basic functions 56

5.1 Sign algebra 70
5.2 Sign algebra of deviations 95

6.1 Execution trace of the Automated procedure 116

7.1 Summarized results of one of the train crossing's scenarios 136
7.2 Scenario produced for TCAS II barometric altimeter deviation 145
7.3 Search trace of Automated on TCAS II example 148
7.4 Search trace (continued) 149
7.5 Search trace (continued) 150
7.6 TCAS II causality diagram node meanings 152
7.7 Node meanings (continued) 153

Acknowledgement

I am indebted to Professor Nancy G. Leveson for the many ways that she has supported my efforts. Her insights and encouragement have been truly inspirational for me. I would also like to thank the other members of my committee, Professors John King and Debra J. Richardson. Clark Turner and Kurt Partridge provided valuable comments on drafts of this dissertation.

I am deeply grateful for the financial support received from the National Aeronautics and Space Administration, the Federal Aviation Administration, and the National Science Foundation.

Mom, Dad, Scott, and Merri, you are the reason I care. Thanks, y'all!

Curriculum Vitae

1989 B.A. in Computer Science and Linguistics,
Rice University
Houston, Texas

1991 M.S. in Information and Computer Science,
University of California
Irvine, California

1995 Ph.D. in Information and Computer Science,
University of California
Irvine, California
Dissertation: *Software Deviation Analysis*

Publications

Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering,* SE-20(9), September 1994.

Steven B. Dolins and Jon D. Reese. A Curve Interpretation and Diagnostic Technique for Industrial Processes. *IEEE Transactions on Industry Applications,* 28(1), January/February 1992.

N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, J.D. Reese, and R. Ortega. Experiences Using Statecharts for a System Requirements Specification. *Sixth International Workshop on Software Specification and Design,* Como, Italy, October 1991.

Abstract of the Dissertation

Software Deviation Analysis

by

Jon Damon Reese

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1996

Professor John King, Chair

An important step in developing safe software is to perform hazard analyses. This dissertation presents an algorithm that, given a formal requirements specification, automatically generates perturbations in the system, which it propagates forward to find potential hazards. In order to achieve this goal, a primitive language of causal dependency is defined. This language represents causal information between system variables directly. Additionally, a qualitative calculus of deviations is presented for calculating normal and deviant behavior. A semi-automated and fully-automated algorithm are defined in terms of these concepts.

Chapter 1

Introduction

Major technological advances in the past century have facilitated inventions of profound usefulness to humankind. We are able to produce more food, fight disease, guard against natural disasters, and otherwise improve our quality of life. However, the same inventions can also be the cause of diseases, disasters, and a degraded quality of life. Bridges and buildings protect us from a harsh environment, but on occasion they fall. Power plants of all varieties heat our homes in the winter and give electricity to hospitals. But power plants, especially nuclear power plants, are capable of large-scale destruction. Even a non-destructive black-out can potentially contribute to the loss of lives. Such technologies as aviation, railway systems, nuclear energy, and medical systems have had a steadily increasing impact on our safety and welfare.

Much of the good and bad that technology can bring is due to an ever-increasing complexity in the things we build. Even as we create tools at the limit of our understanding, we use them to design even more complicated systems. This trend is not limited to advanced research; systems are routinely developed with increasing complexity [2]. Systems have likewise increased steadily and significantly in sheer size [31]. Size and complexity make the task of carefully investigating the hazards to which a device or process may expose its environment more difficult [31]. This,

combined with the difficulty operators have in understanding hazards as they occur, and especially in bringing them under control, compromises our ability to design safe systems. As Charles Perrow observes:

> In the last fifty years, ... and particularly in the last twenty-five, to the usual cause of accidents — some component failure, which could be prevented in the future — was added a new cause: interactive complexity in the presence of tight coupling, producing a system accident [26].

The means of system control influences complexity. This is especially true of automatic controllers; the potential "coupling" that Perrow mentions above is limited if the control loop requires actions from a human operator. Historically, systems have been controlled by mechanical or electrical means, although pneumatic, hydraulic, and other devices are also commonly placed in the control loop. However, computers and the software they contain are steadily replacing many of these conventional technologies. For example, in 1955 only ten percent of U.S. weapon systems contained software. By 1981, more than 80 percent were computerized [15].

The reasons for computerization are many-fold, though not all of them are sound. Computers can centralize process control, a common but debatable goal. Software is relatively easy to modify, saving time and expense for the developers. It is more precise and accurate than many conventional technologies. Computers perform millions of instructions per second, making software the fastest alternative to perform complex tasks. Additionally, the combination of computer screen and flexible operating modes makes the computer an attractive alternative to the "traditional operator's control panels, festooned with switches, buttons, and lamps." Unfortunately, the computer is also used as a marketing tool; "computerized" systems are often perceived by industry, the government, and the general public as uniformly superior to older technologies. Finally, computers are used to improve system safety.

The arguments given here comprise only a partial list of why computers are increasingly prevalent in safety-critical systems, but it is strong argument that software has a growing role in process control. However, software is certainly not without its problems. First and foremost, it increases the very complexity it seeks to treat [20]. This is at least partly due to the aforementioned phenomenon of attempting to build ever more complex systems as they become possible:

> [T]he availability of enormous computing power at a low cost has led to expanded use of digital computers in current applications and their introduction into many new applications. Thus larger and more complex systems are being designed. [4]

The additional complexity is compounded by the nature of control software. While conventional technologies exhibit typically continuous behaviors (consider, for example, the sawtooth-shaped voltage of an overdriven capacitor or the deflection of a stressed beam) software typically exhibits discrete reactions to its inputs. This behavior can be dangerous since small perturbations of software inputs often result in huge deviations in the outputs [15, 26].[1] Additionally, the mathematics governing programming languages are not as well understood, nor as easily calculated, as those for analog systems [15].

Software engineering is a relatively new discipline, and there is very little historical data on its use as compared to other fields of engineering. The products of software have very unpredictable failure modes and rates compared to, say, metallurgy, solid state electronics, or chemistry. Although more will be learned as software engineering matures, efforts are exasperated by the wide range of possible behaviors

[1]Devices other than digital computers are certainly capable of such reactive behavior, *e.g.*, a switch connected to an explosives detonator. However, it is common-place for there to be a highly non-linear relationship between software inputs and outputs.

enabled by software. Process-control software is especially difficult to characterize, as designs generally result in "frontier software," which tests the limits of programming practice [24].

As we place more faith in computers, we become more vulnerable to their failures. Peter Neumann has compiled a list of over 400 cases in which computers contributed to an accident or near-accident, some of which resulted in loss of life. Some incidents are due to computer hardware failures. However, the majority are due to some sort of software error [8]. Computers have contributed to an increasing number of medical equipment recalls by the U.S. Food and Drug Administration. For example, the number of recalls doubled between the years of 1982 and 1984 [13]. According to Forrester and Morrison [8], failures are common in telephone switching software, air traffic control systems, bank automated teller machines, electronic funds transfer systems, industrial robots, and police computers.

One proposed solution to the problem of unsafe software is to remove the computer from safety-critical systems altogether [8]. Similarly, the problem may sometimes be avoided by not building the system at all. These options should seriously be considered; however, it is unrealistic to expect such rules to be applied categorically. Computer software has many strengths, and sometimes it is the best alternative. Regardless, computers will continue to be installed in safety-critical devices, and it is the responsibility of software and system engineers to develop methods of making software safer.

Unit, integration, and operational testing are the traditional ways to make software more reliable. While testing is a necessary part of software development, in general it is not sufficient for safety-critical software. Some operational tests cannot be performed at all, *e.g.*, inducing a meltdown in a nuclear reactor or firing missiles

at the United States to test the SDI system proposed in the 1980's. Operational conditions can be simulated during integration testing, but assumptions about the environment may be wrong. Even if operational testing can be performed, there are probably insufficient resources (including time) to obtain a high enough statistical confidence in the software. For example, "ultra-reliable" software with an expected failure rate of 10^{-10} per hour would require approximately **114** years testing of 10,000 replicates [4]. About one-third of all software errors do not appear until after 5,000 operation years [20]. These figures do not include the added burden of testing in the presence of failures of other components. It is difficult, if not impossible, to test the software under all failure modes of the system. Thus, although testing is a necessary part of protecting the system against unsafe software, other steps must be taken to insure system safety.

1.1 Contribution of the dissertation

An important step in developing safe software is to perform hazard analyses. Hazard analyses were first developed in the fields of nuclear power and weaponry, aviation, and space technology [31]. As is shown in chapter 2, there are a variety of hazard analysis procedures, but few can handle the complexities of software. In particular, there is a lack of forward search methods for the analysis of software requirements.

This dissertation presents an algorithm that, given a formal requirements specification, automatically generates perturbations in the system, which it propagates forward to find potential hazards. The algorithm can be used to explore how deviations in the system can affect the controller's behavior (and, via feedback, the

process's state) and how deviations in the controller's behavior can affect the process. In order to achieve this goal, a primitive language of causal dependency will be presented. Additionally, a framework is provided for calculating normal and deviant behavior at varying levels of detail. The algorithm is defined in terms of these concepts.

1.2 Organization of the dissertation

This dissertation is organized into eight chapters, including this introduction. Chapter 2 is a survey of existing hazard analysis procedures. The contributions of this thesis are introduced in chapter 3. These contributions are then each treated in detail in the following chapters. A new primitive language of causality is presented in chapter 4. Chapter 5 provides a theoretical and practical framework for the development of a calculus of deviations. These concepts form the basis for a forward analysis algorithm developed in chapter 6. Chapter 7 presents examples. Chapter 8 summarizes the results and suggests new research areas for deviation analysis.

Chapter 2

Related Work

This chapter presents hazard analysis techniques that may be considered to address software safety concerns. These methods are drawn from academic discourses in software and system safety and from the state of practice in various industries. The present survey is not intended to be complete with respect to these fields of knowledge. Rather, evaluation of the methods is limited to applicability to requirements of embedded-control software.

Given a system and an anticipated operating environment, *hazard analysis* is the methodical investigation of the system, in part or as a whole, for potential hazards. The concept of a *hazard* has been defined in a variety of ways. A traditional engineering definition is the "potential for an uncontrolled transfer of energy having the capacity to result in such undesired effects as death and injury."[31] This definition precludes software from being in a hazardous state since it does not directly involve an uncontrolled transfer of energy. Even if the definition is interpreted to include indirect releases of energy, it excludes other dangerous situations. For example, a non-operational patient monitoring system can lead to an accident in which no energy is released, controlled or otherwise.

Another definition of hazard, suggested by [30], is a "peril, danger, or risk." While this captures the intuitive notion of a hazard, and certainly cannot be proven

to exclude any cases, it is too ambiguous to serve as a technical definition. Probably the most useful definition is that of [16]:

> a *hazard* is a state or set of conditions of a system ... that together with other conditions in the environment of the system ... will lead to an accident (loss event).

Thus, a hazard is a state of the system such that an accident is inevitable under certain circumstances. Note that this definition differentiates between the system and its environment. A useful working definition of the system is the controller and the part of its environment over which it has some control [16] (thus dividing the controller's environment into "system" and "environment.") For example, consider an autopilot on an aircraft. The autopilot is the controller. It exercises at least partial control over the entire aircraft, but not over anything external to the aircraft. Therefore, the autopilot system is defined by the borders of the aircraft. The operating environment is everything else that the aircraft encounters, including weather, other aircraft, the ground, and radio signals. A potential hazard is a downward-drifting autopilot in an airborne aircraft. Under some circumstances (*e.g.*, over foggy, mountainous terrain) an accident will occur. Note that the hazard is not present when the aircraft is grounded. A hazard description must include the state of the system and must be defined with respect to some (realistic) environmental conditions.

Hazard analyses may conceivably be performed at any stage of system development. However, there is evidence that the earliest stages of development may benefit most from a hazard analysis. In a study limited to the causes of safety-related software errors [21], Lutz found that the errors that persist until integration and system testing are usually due to difficulties with the software requirements. In particular, she reports that "safety-related functional faults are more likely than non-safety-related functional faults to be caused by requirements which have not been identified." Thus,

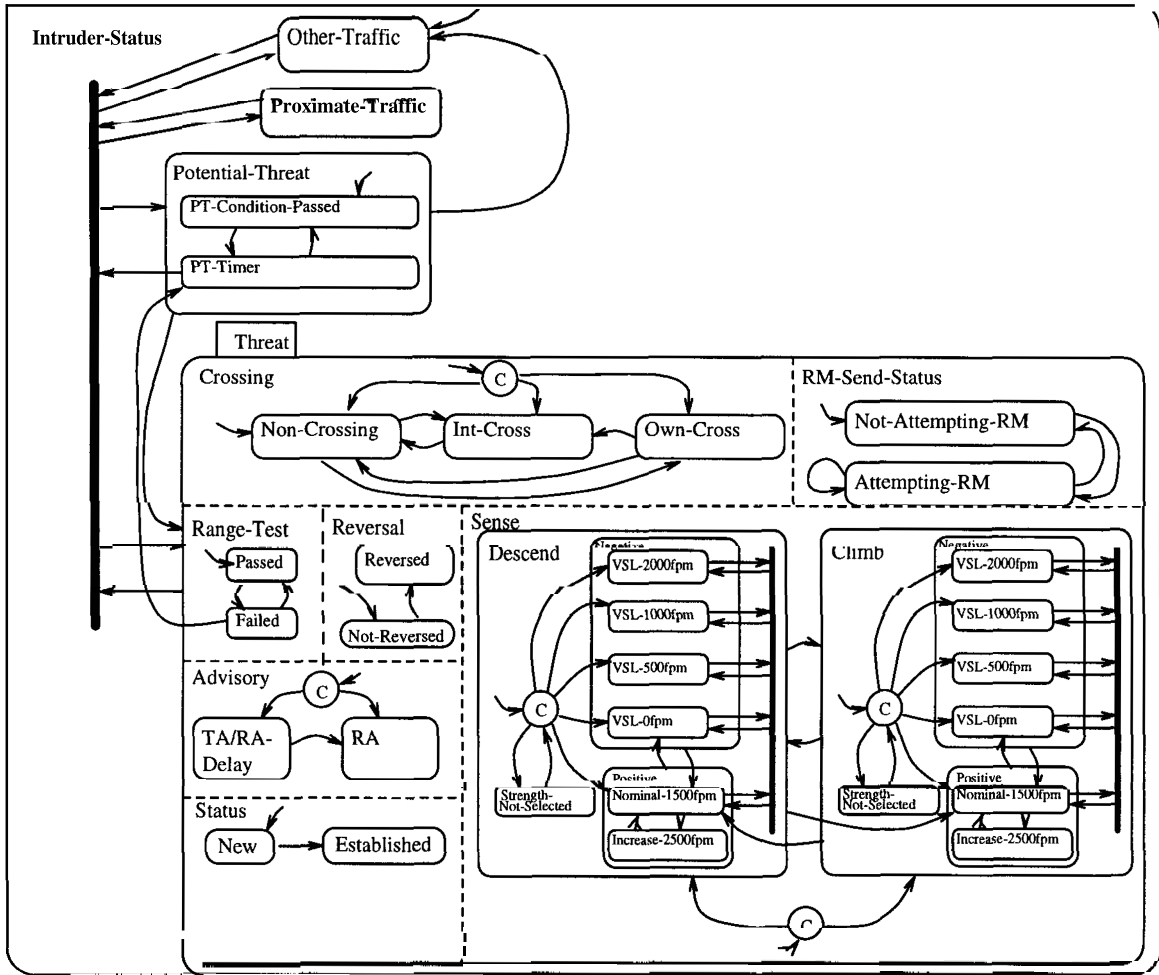


Figure 2.1: The TCAS II model of detected aircraft, written in RSML [6].

one may conclude that it is most beneficial and easiest to perform analysis early in the design process.

A system may be represented in two complementary ways. A *structural* definition represents the system as a static hierarchy of elements. Structural relationships are invariant with respect to time. For example, the RSML diagram in figure 2.1 contains a variety of structural definitions. The states *Other-Traffic*, *Proximate-Traffic*, *Potential-Threat,* and *Threat* are mutually exclusive. This relationship is independent of time; if one state is active then the others must be inactive. The state *Threat* is coexistent with its sub-states: *Crossing, RIM-Send-Status, Range-Test, etc.*. If one of

this group is active, then all of them are. Again, this relationship may be applied without knowledge of past activity.

RSML states are boolean—active or inactive. Structural definitions may take other mathematical forms. For example, the ideal gas law, $PV = nRT$, is invariant with respect to time.

The dynamic behavior of a system may be represented as a mapping from state to subsequent state. *Dynamic* definitions differ from structural definitions in that the relationships depend on time. The transitions in the *Intruder-Status* example transitions are dynamic definitions. For example, if state *Potential-Threat* is active, then the transition to *Other-Traffic* defines how *Intruder-Status* can be in that state in the next instant. (Note that the conditions for taking the transition are not shown in the diagram.)

Most hazard analysis techniques involve a search of some sort. The type of search may be categorized with respect to the direction of analysis. “Forward” searches are those in which the analyst traces from cause to effect. If one conceives of the system as a function, then a forward search works from domain to range. In contrast, “backward” searches work from range to domain, thus representing a trace of the function’s inverse relation.

The techniques presented in this chapter are divided into backward searches, forward searches, and combined approaches.

2.1 Backward Search Techniques

2.1.1 State Machine Hazard Analysis

State Machine Hazard Analysis (SMHA) is an algorithmic procedure developed specifically for the analysis of software requirements [18], although the method can be adapted to any stage of design [16]. SMHA was originally defined for a timed Petri net language [18]. It has been adapted to Statecharts [23, 24] and an effort is underway to adapt it to RSML. The algorithm takes as input a model of the system described in a formal, state-based language. The algorithm also requires the system state space to be partitioned into complementary "hazardous" and "safe" states. The algorithm also supports analysis of safety in the presence of failures if the formal language supports the specification of "failure" transitions (described below).

The algorithm begins with a set of hazardous states that the analyst wishes to inspect. It chooses one and does a search backward along all paths leading to that state. The search terminates at a "critical" state — a state that is safe and leads directly to another safe state (other than the one that is being followed backward, if it happens to be safe.) The critical state is the last point at which the system can avoid a hazard. The requirements must be changed to eliminate the undesired transition from the critical state. The algorithm then picks another hazardous state to inspect.

Note that the algorithm is conservative in that it requires the removal of bad transitions at all critical states, whether or not the critical state is reachable from the initial state. The analyst is forced to remove hazardous states that may not be reachable, but the added burden is probably far outweighed by the savings in analysis efforts in determining whether a critical state is reachable. Thus, SMHA

is a labor-efficient method of hazard analysis. Additionally, designing in these extra safety constraints provides a measure of protection against implementation errors and hardware failures.

One consideration of the algorithm is whether any critical states could lead eventually only to hazardous states, since the safe alternative may itself lead only to hazardous states. This problem is avoided by having the hazardous transitions removed as they are found. This is accomplished by adding the transition to a list for the analyst to inspect and removing it for the reachability graph. In this way, the next search to reach a particular critical state has one less transition to choose from. By the time the last hazardous state is analyzed, that "critical" state is no longer critical, because all transitions but the one under inspection have been eliminated from the reachability graph; the search continues backward to the new critical state. Thus the algorithm facilitates the removal of the hazards under inspection.

SMHA must be performed after the hazard identification phase since it takes the hazardous states as input. Like any backward analysis method, it is most suited for a relatively small number of hazardous states [18].

2.1.2 Fault Tree Analysis

Fault tree analysis (FTA) was developed by Bell Laboratories in 1961 for the Minuteman missile project [14]. The analysis begins with the identification of some undesired event, called the "top event." The analyst determines the conditions that could comprise the top event and places them in a graphical layout similar to that shown in figure 2.2. If a single condition in a group of conditions is sufficient to cause

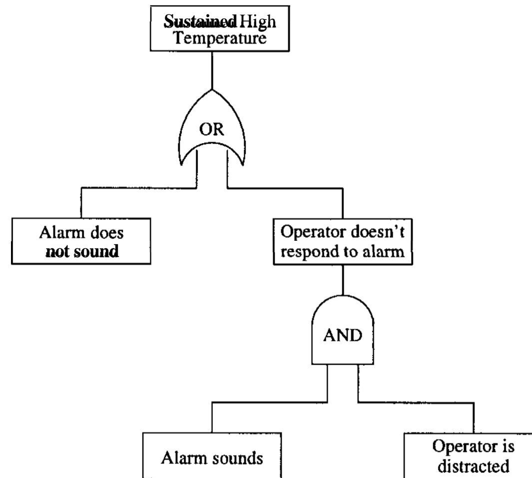


Figure 2.2: Fault tree.

the top event, then the conditions are all connected by an OR gate; an AND gate is used if all conditions must be true for the top event to occur.

Treating each condition as a new top event, the analyst proceeds successively to build a tree in which causality can be traced from the leaves to the root. The tree is finished when all the leaves are "primal events," which are considered to occur stochastically or are otherwise resistant to a logical analysis. The decision of when to stop is at the discretion of the analyst [16], subject to the information available.

Fault trees may be used qualitatively to determine whether or how a particular top event is possible by inspecting "cut sets." A cut set is a set of primal events such that all of the events are necessary and sufficient to cause the top event [14]. Algorithms exist to produce a fault tree's cut sets automatically. Thus the fault tree is very useful in determining the causative relationship between primal events and the top event [14].

FTA is traditionally a probabilistic methodology [7]. A quantitative analysis can be performed if probabilities are provided for the primal events, although the

analysis is complicated by non-independent events. However, the validity of probabilistic methods has been called into question. Actual rates of failure in practice have exceeded calculated values by several orders of magnitude [27]. Probability densities are usually ignored [30]. Consequently, no statistical confidence can be attached to the top event's probability, severely restricting its usefulness. Especially with respect to software, numbers can be very difficult to obtain. In order to complete a probabilistic analysis, the analyst must assign arbitrary values for software failures. In light of this dilemma, the FAA specifically excludes software from any quantitative analysis requirements [20].

FTA is used extensively in safety programs in the nuclear power and weapons industries. It has also gained widespread acceptance in other industries.[1]

Although FTA has proven useful in practice, manually-constructed fault trees suffer from several weaknesses. They can require much time and effort to construct [5, 9, 14, 33]. Also, they are subject to logical errors and omissions [14]. Because of this, and because FTA "is an art, rather than a science" [33], different analysts often produce fault trees that are inconsistent with each other [5, 15, 33].

Several algorithms have been developed to alleviate problems with the manual construction of fault trees. One family of algorithms is based on the technique of "mini-fault trees." The other family of algorithms uses "digraphs."

[1] A variation of FTA is the Management Oversight and Risk Tree (MORT). MORT is guided by a 1500-item questionnaire related to system management, human behavior, and environmental factors [16]. These issues are outside the scope of this dissertation.

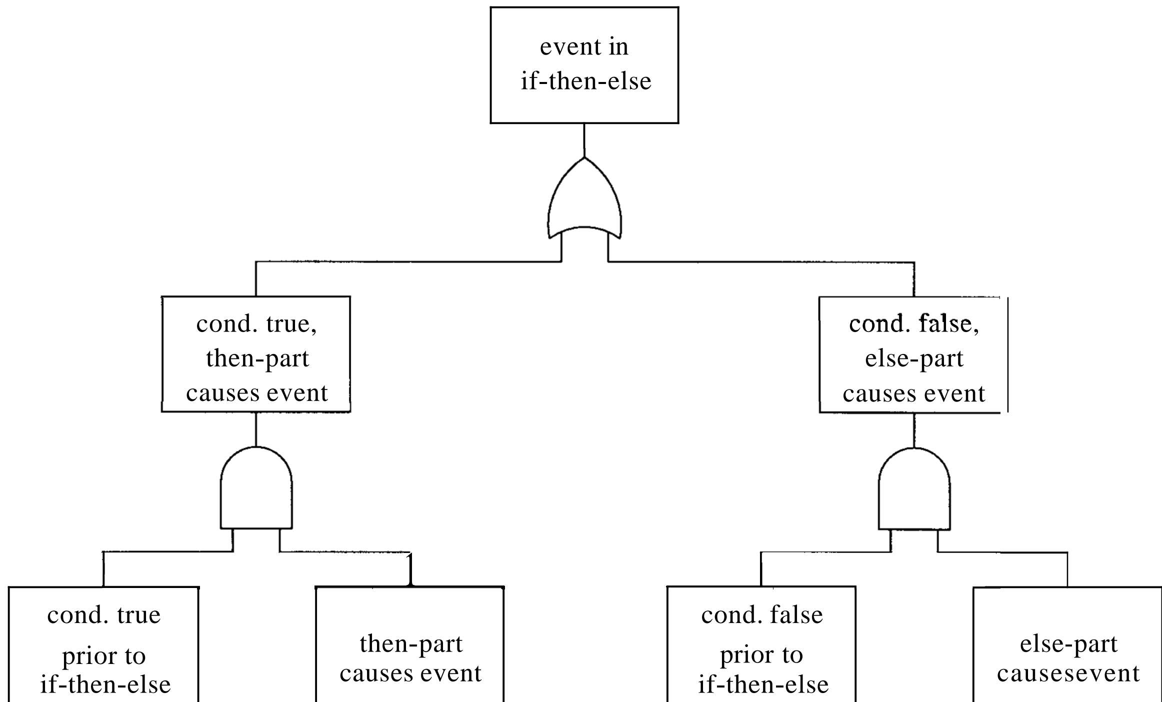


Figure 2.3: Mini-fault tree for IF-THEN-ELSE. Taken from [17].

Mini-Fault Trees

The term "mini-fault tree" was coined by Taylor and the concept was developed independently by Taylor and by Leveson in the early 1980's. A mini-fault tree is a fault tree fragment for a particular language construct, such as a programming language statement (see figure 2.3 for an example). The top node of a mini-fault tree is an event to expand. The leaves of a mini-fault tree are events or conditions that must be investigated in order to show that the top node can or cannot occur. For each leaf, the analyst has the option of attaching another mini-fault tree or expanding it by hand.

In practice mini-fault trees are usually equivalent to the application of weakest pre-condition rules. The weakest precondition $wp(S, R)$ is defined as the weakest logical statement about the state in which execution of statement S results in state

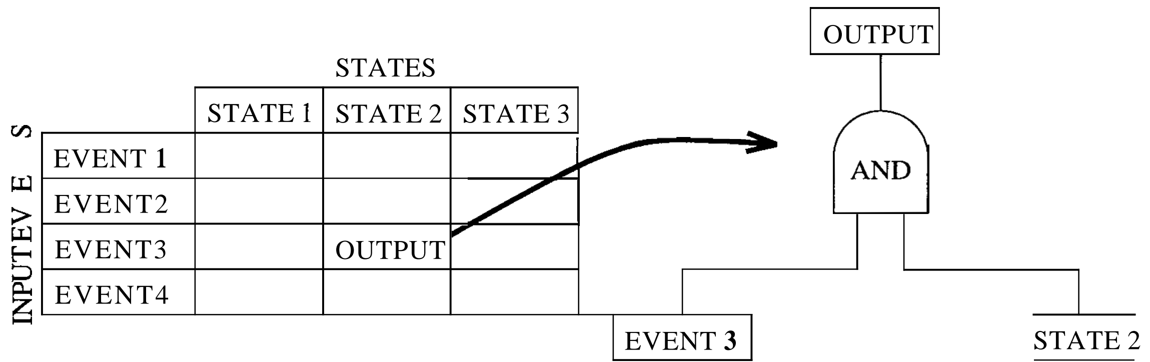


Figure 2.4: Mini-fault tree constructed from standard component transition table. The output is either an event or a new state.

R. The root node of a mini-fault tree is equivalent to state R occurring as a result of statement S. The remaining nodes define the weakest precondition. For example, the mini-fault tree in figure 2.3 is equivalent to the following weakest pre-condition rule:

$$\begin{aligned} wp(\texttt{if-then-else}, event) = & (\texttt{cond.} \wedge wp(\texttt{then-part}, event)) \vee \\ & (\neg\texttt{cond.} \wedge wp(\texttt{else-part}, event)) \end{aligned}$$

Mini-fault tree analysis methods have been outlined for both hardware and software. Taylor describes an algorithm based on a process flow sheet of the plant, such as a piping and instrumentation diagram [33]. The system is assumed to be a set of standard components, such as valves and pumps, inter-connected by ports. Each port has process variables defined for it, such as pressure and temperature. A *state transition table* is defined for each standard component mapping an input variable event ("pressure becomes high") and a component state to a new state or output event. The mini-fault trees are constructed from these tables. See figure 2.4 for an example.

Leveson and Harvey [17] define software fault tree analysis (SFTA). The goal of SFTA is either to prove that the software cannot cause a particular event or to show the circumstances under which the the event can occur. A standard fault tree

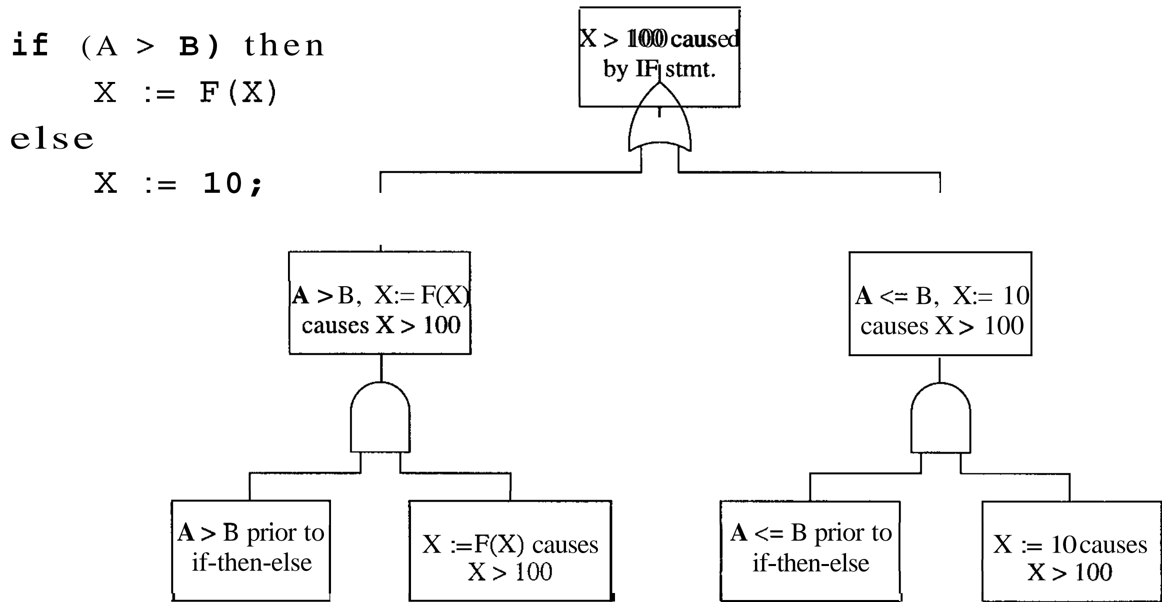


Figure 2.5: An instantiated mini-fault tree (taken from [17].)

is constructed until a leaf node involves an assertion on the result of a particular software statement. The analysis tool then attaches the program statement's mini-fault tree onto the fault tree at the appropriate place, substituting appropriate values (see figure 2.5 for an example.)

Software fault tree analysis is semi-automated. Leaves of a mini-fault tree that reference other statements can have their mini-fault trees appended (*e.g.*, (‘then-part causes event” and ‘(else-part causes event” in figure 2.3.) This activity is equivalent to the expansion of $wp(S, R)$ terms in a weakest pre-condition definition.

The SFTA templates have been defined for the programming language Ada, and hence for most algorithmic programming language statements. It has been used extensively, including for shutdown software at Ontario Hydro nuclear reactor [3] and the University of California, Berkeley FIREWHEEL spacecraft [17]. The methodology has also been extended to the Statecharts specification language [23, 24], showing

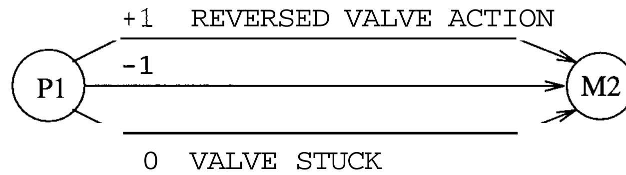


Figure 2.6: A simple digraph.

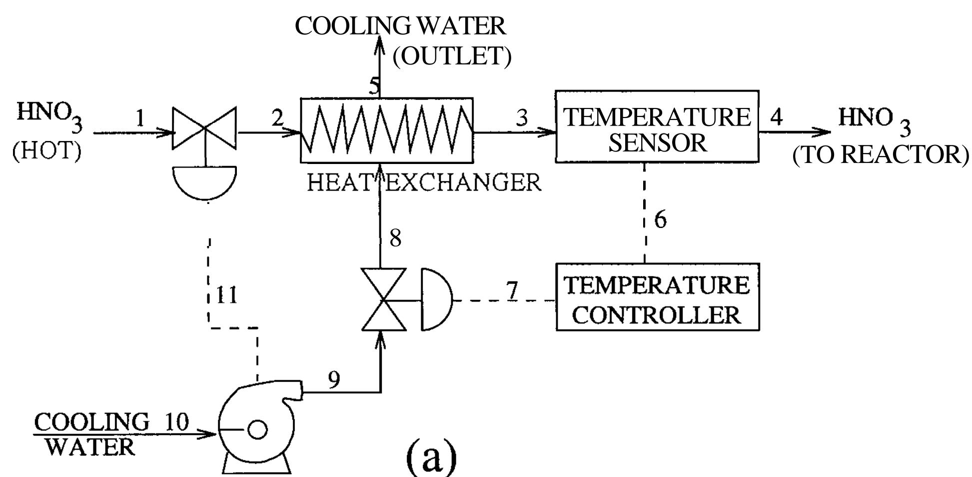
the applicability of mini-fault tree analysis, and especially SFTA, to the analysis of software requirements.

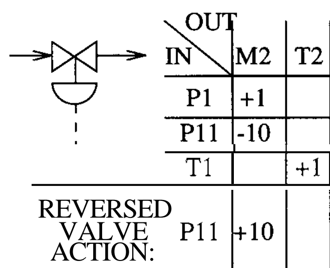
Digraphs

The other method of automating fault tree analysis is the use of directed graphs, referred to as *digraphs* in the literature. Digraphs are used in failure analysis to construct a model of failure propagation. Nodes in a digraph represent process variables. The directed edges represent lines of influence between variables. Figure 2.6 illustrates a simple digraph.

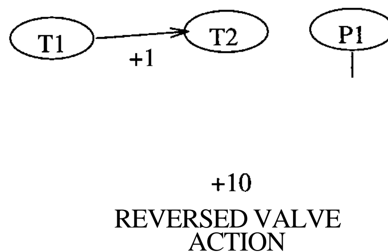
The basic procedure for producing a fault-tree using a digraph is to first convert the system description into a digraph and then traverse the digraph backward (*i.e.*, in the opposite direction of the edges.) The conversion of specification to digraph can be automated if standard components are used [14]. For example, figure 2.7(a) shows a simple pipe-and-process diagram. Figure 2.7(b) is a description of the standard failure behavior of one of its components, a valve. Figure 2.7(c) shows a portion of the resulting digraph based on the failure behavior in (b).

Digraphs can be augmented by specifying the type of influence a failed variable can have on another variable. Lapp and Powers [14] allow the set of values $\{-10, -1, 0, +1, +10\}$ to appear on digraph edges. These values are simplifications of the partial derivative of the destination variable's value with respect to the source





OUT / IN	M2	T2
P1	+1	
P11	-10	
T1		+1
REVERSED VALVE ACTION: P11	+10	



(b) (c)

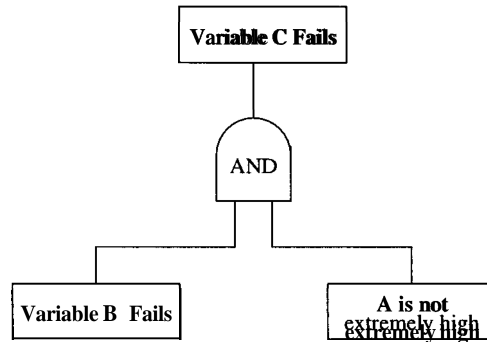
Figure 2.7: (a) A simple pipe-and-process diagram. (b) Failure description of a valve. (c) The digraph resulting from applying the failure description (b) to figure (a).

variable. A value of ± 1 indicates that a failure in one variable causes another to deviate from normal value by a moderate amount. A value of ± 10 indicates that a very large deviation results. A value of 0 indicates that the source variable has lost influence over the controlled variable. The sign of the edge represents the direction of deviation in the controlled variable. The Lapp and Powers method is an example of the use of qualitative mathematics, which is discussed in chapter 5.

A digraph can include multiple edges between two nodes. In this case, all but one of the edges are qualified with failure conditions. The condition may be considered external to the model (as all conditions are in the Lapp and Powers version) or it may be a predicate of other variable values, as in the logic flowgraph [9].

The basic algorithm for constructing a fault tree begins with an assumed failure in one of the process variables (*i.e.*, at one of the nodes in the digraph.) If the digraph is augmented with values, then a specific failure is identified (such as a value of $+10$.) This is the top event of the fault tree. The algorithm inspects all edges leading into the digraph node and constructs an *OR* gate under the top event consisting of all the variable failures that could lead to the top event. Then each of the failures is inspected in turn. If several failures must act in conjunction to produce the top event, then those failures are placed under an *AND* gate.

The algorithm must consider cancelling effects by other variables. For example, if an extremely high value for variable A can cancel variable B's effect on variable C, then a treatment of variable B's influence must include the negation of the possible influence of variable A:



Digraphs have not been adapted successfully to software. In particular, the use of (constant-valued) partial derivatives as a basis for influence is inadequate for expressing the often complex relationship between software inputs and outputs. However, the use of qualitative values for variable deviations appears to be a promising approach, as will be explained in subsequent chapters.

With the exception of logic flowgraphs, the digraph models do not incorporate binary relationships between variables.

2.2 Forward Search Techniques

Forward search techniques focus on whether or how a normal or failure state can cause a hazard.

2.2.1 Failure Modes, Effects, and Criticality Analysis

Failure modes and effects analysis (FMEA) is employed to determine the effects of single failures on a system's performance [7] (contrasted with fault tree analysis, for example, which allows the consideration of multiple failures.) FMEA was developed

Part name	Potential Failure Mode	Potential effect of failure mode	Risk Priority
Positive input wire to pump	Open circuit	No screenwash	108
	Short to ground	This might (should) blow a fuse	72
	Short to positive	The pump will be permanently on. The water will quickly run out, resulting in no screenwash	27
Input to CPU	Open circuit	No screenwash	108
	Short to ground	The pump will be permanently on. The water will quickly run out, resulting in no screenwash	**54**
	Short to positive	This might blow the tracking on the switch which would result in no screenwash.	**54**
Screenwash pump	Stalled pump	The pump has a stall current of 10 **A.** The fuse is rated at **5 A.** The fuse will therefore blow.	72
Negative input to relay	Open circuit	No screenwash	108
	Short to ground	The pump will be permanently on. The water will quickly run out, resulting in no screenwash	**64**
	Short to positive	This would blow the CPU Darlington output	**54**

Figure 2.8: Example of a Failure Modes, Effects, and Criticality Analysis (FMECA) table. The second column lists failures modes, the third column failure effects, and the last column lists criticality ratings. Taken from [28].

as an aid to reliability analysis, but a variant known as Failure Modes, Effects and Criticality analysis (FMECA) has been used to identify potential hazards [16].

FMEA is basically an ad-hoc procedure based on a tabular form (see figure 2.8 for an example.) The first step of the procedure is to list all of the components of the system. Next the analyst lists all of the possible failure modes and failure rates for each component. All these data are entered into the FMEA forms. The analyst then determines and describes all the possible effects for each failure [16]. For a FMECA this information includes hazard severity, the likelihood of detection, and frequency of occurrence. If a failure rate is sufficiently high and the effect is sufficiently serious,

then the system must be redesigned [25]. A search for failure causes is not a part of the FMEA procedure [28].

FMEA has been used extensively and is well-understood by industry [7]. It is most appropriate as a detailed analysis of a single, standard component [16].

FMEA is described as a slow and tedious procedure [7, 25, 28]. There are no tools to lessen the analyst's burden of investigating many similar effects, and there has been little research into the automation of FMEA [28]. The amount of labor involved causes FMEA's to be expensive, since they must be performed by experts [25]. In particular, the method leaves to the analyst the burden of investigating many similar failure patterns [28] (or recognizing that such patterns exist.)

Another weakness of FMEA is its concentration on single failures. In contrast to methods like HAZOP, they do not treat hazards arising in component interfaces [16]. Finally, FMEA's lack of structure with respect to identifying failure modes or searching for effects is problematic for a software hazard analysis. As discussed in chapter 1, control software behavior is often complex and disjointed. Analysts can be overwhelmed by the multitude of variable interactions. FMEA does not provide any assistance in structuring a search for failure effects.

2.2.2 Event Tree Analysis

Event Tree Analysis (ETA) was developed as a hazard analysis method for the nuclear industry. The basic structure of ETA is a decision tree, called an *event tree*. Each node in an event tree represents the occurrence or absence of an event, usually a failure. The event tree is constrained in two ways. All nodes that are equidistant from the root node address the same event. Also, time must proceed monotonically

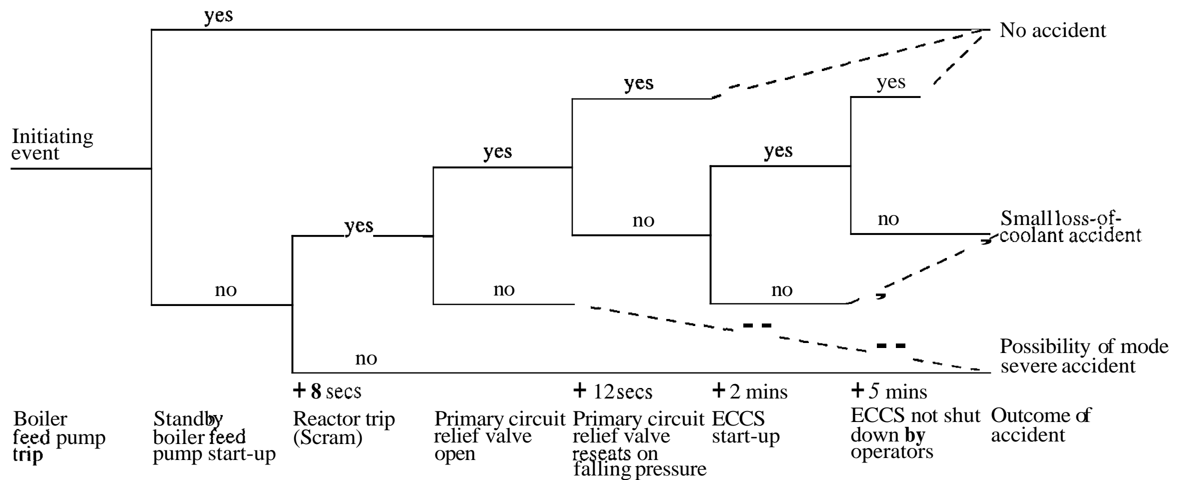


Figure 2.9: An event tree, taken from [16].

down the tree, *i.e.*, an event node cannot occur before its parent. Figure 2.9 shows an example event tree.

ETA is most commonly used for probabilistic analysis. Each event is assigned a numeric probability. The probability of a hazard occurring via a particular path is the product of all of the event probabilities along that path, assuming the events are independent. The total probability of a particular hazard occurring is the sum of each path leading to that hazard. In the example in figure 2.9, a boiler feed pump trip may lead to a severe accident if the standby boiler feed pump fails to start and the reactor fails to trip or if the standby boiler feed fails to start, the reactor trips and the primary circuit relief valve does not open. Given the following fictitious probabilities for each event

$$\begin{aligned}
\text{Standby boiler feed pump starts} &= P(a) = .99 \\
\text{Reactor trip (Scram)} &= P(b) = .98 \\
\text{Primary circuit relief valve open} &= P(c) = .97
\end{aligned}$$

the probability of a severe accident is

$$\begin{aligned}
&= [1 - P(a)][1 - P(b)] + [1 - P(a)]P(b)[1 - P(c)] \\
&= (.01)(.02) + (.01)(.98)(.03)
\end{aligned}$$

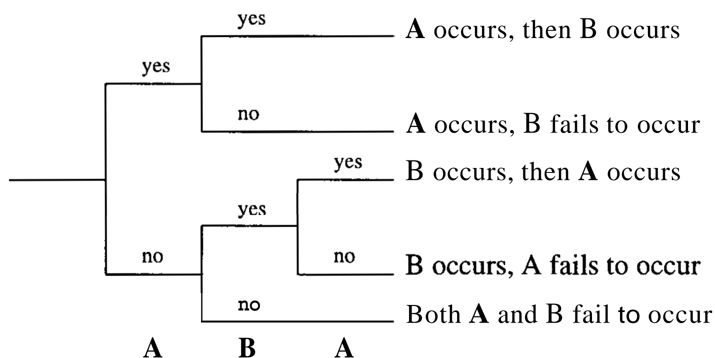


Figure 2.10: An event tree for two unordered events A and B.

$$= 2.00 \times 10^{-4} + 2.94 \times 10^{-4}$$

$$= 4.94 \times 10^{-4}$$

This equation assumes, of course, that $P(a)$, $P(b)$, and $P(c)$ are independent.

ETA is useful for presenting in detail the possible scenarios that can transpire in the event of multiple failures. However, the thoroughness comes at great cost. The trees can require quite a bit of space on the page, due to the need to list all events across the bottom of the page. The list of events can also grow quite long if events do not have a strict sequence. If events A and B may occur as AB or BA, then the event tree must have ABA at the bottom of the page to accomodate the alternative sequences (see figure 2.10.) Both of these problems are mitigated by separating the tree into sub-trees, each of which can have its own list of relevent events. The relationship between a sequence of events and the resulting failure is not explicitly shown. ETA is limited to a single initiating event. Finally, and most seriously, the probabilistic analysis becomes quite complicated if events are not independent. Unfortunately, it is difficult to prove (and dangerous to assume) that two events are independent.

ETA is appropriate only at the detailed design stage and afterward [16]. There do not appear to be any automated methods for constructing an event tree, although ETA appears to lend itself to automation as well as manual analysis.

2.3 Combined Techniques

Some hazard analysis techniques combine forward and backward searches by beginning with a scenario, searching backward to find ways in which the scenario can occur, and searching forward to determine whether the scenario leads to a hazard. Methods that employ such a strategy may be considered "exploratory" since neither hazards nor their causes are necessarily known beforehand.

2.3.1 Cause-Consequence Analysis

Cause-consequence analysis (CCA) is a semi-automated hazard analysis technique that combines forward and backward searches. The source of analysis is a block diagram of the system, where each block represents a functional unit and a line between blocks represents the output of one block serving as input to the other.

CCA produces a cause-consequence diagram based on the system description. Figure 2.11 shows an example of a cause-consequence diagram. An *event box* describes a change in the system state. A *decision boa:* shows alternative effects of an event, depending on a *condition box* or *condition tree.* Condition trees resemble fault trees, except that the top event is the condition that makes the decision box true. A condition box is a single-node condition tree. The notation also provides for inhibitory relationships between events. This symbol is useful to show mutual exclusion, *i.e.*, to

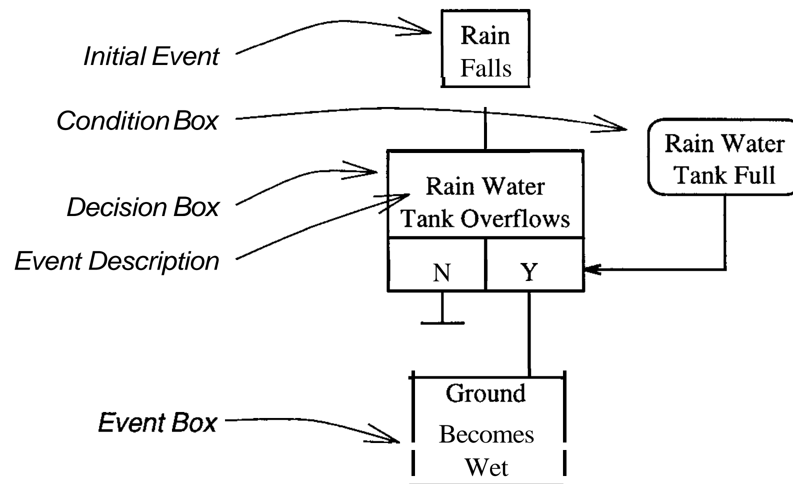


Figure 2.11: A cause-consequence diagram, taken from [32]

show that an event can cause exactly one consequent event of multiple alternatives. All the symbols except for the condition boxes and trees form a *decision graph.*

The algorithm begins with an initial event. It traces the event backward until events can be considered spontaneous. The procedure is essentially the same as constructing a fault tree, suggesting that the algorithms described earlier are applicable here. The algorithm then traces forward to find the effects, constructing the decision graph.

As a manual technique, CCA is a systematic analysis of system block diagrams that results in a notation in which sequence is shown explicitly [32]. As a semi-automated technique, it has several shortcomings. The analyst must decide when a particular search path will not lead to a hazard. Also, contrasted with HAZOP (presented in the next section) the algorithm does not automatically identify failure modes. This lack is not an important flaw in a highly heterogeneous process, where behavior is not easily specified anyway, but CCA is not as useful for software process control, where system behavior is already abstracted and specified. Thus, it would be much more helpful to the analyst for the analysis procedure to postulate failure

modes automatically, rather than relying on expert input. More seriously, CCA cannot represent feedback directly. All components involved in a feedback loop must be collected into a single component for purposes of analysis. Finally, the diagrams can become "unwieldy." [16]

2.3.2 Hazard and Operability Study

Hazard and operability analysis (HAZOP) is a semi-formal review procedure developed for the chemical industry to cope with potential hazards and other disturbances in operations. The goal of a HAZOP is to identify operational deviations from intended performance and study their impact on the system's safety [31]. The HAZOP procedure is carried out by a HAZOP expert (the leader) and a team of system experts. The leader poses a battery of questions to the experts in an attempt to elicit potential system hazards. A HAZOP is basically an exploratory analysis, as neither potential faults nor hazards have been identified beforehand [22]. Rather, the HAZOP leader hypothesizes an abnormal condition and analysis proceeds in both directions determining whether and how the condition is possible and what effects it has on the system.

The analysis follows a systems theory model of accidents [16], in that it concentrates on the hazards that can result from component interaction, *i.e.*, accidents are caused by deviations in component behavior. The basic document that a HAZOP draws from is a pipe-and-process diagram. Each pipe has certain *process parameters*, such as pressure, temperature, and chemical composition. A list of guide words is applied to each parameter to yield an inventory of *deviations* from normal or expected behavior. See table **2.1** for a typical list of guide words. An example of a deviation is the guide word "MORE" applied to pipe A's temperature. The analysts are asked

NONE	Intended result is not achieved.
MORE	Too much of a particular parameter.
LESS	Not enough of a parameter.
AS WELL AS	Unintended activity or material.
PART OF	Parts of the parameter are missing.
REVERSE	Parameter's value is opposite of intended value.
OTHER THAN	Something other than the intended result happens.

Table 2.1: HAZOP guide words (adapted from [16].)

the two questions "What is the effect of pipe A's temperature being too high?" and "How can pipe A's temperature get too high?" Since the questions center around pipe parameters, HAZOP has been characterized as a "flow-based" analysis [22].

A deviation that can occur and can lead to a hazard is a *meaningful* deviation [22]. Often, this definition is modified to be probabilistic, *i.e.*, a deviation that can occur with sufficiently high probability and can lead to a hazard with sufficiently high probability.

HAZOP has been used extensively in the chemical, nuclear and food processing industries [22]. The success of HAZOP may be attributed to several factors. The hypothetical nature of HAZOP questions encourages creative thinking [16]. Also, HAZOP studies are typically performed by a team of analysts, led by a HAZOP expert, resulting in a potentially very useful exchange of information and opinions. As a result, the HAZOP procedure is almost uniquely capable of systematically identifying new hazards in a proposed design.

Since the procedure focuses on flow at the exclusion of component functionality, a preliminary HAZOP can be performed early in system design, though the results will likely be quite preliminary.

HAZOP has several limitations. It is time- and labor-intensive [16], in large part due to its reliance on group discussions. HAZOP analyzes causes and effects

OMISSION	Intended output is missing.
COMMISSION	Unintended output is generated.
EARLY	Output is generated sooner than intended.
LATE	Output is generated later than intended.
COARSE INCORRECT	Output's value is wrong.
SUBTLE INCORRECT	Output's value is wrong, but cannot be detected.

Table 2.2: MASCOT guide words.

with respect to deviations from expected behavior, but it does not analyze whether the design, under normal operating conditions, yields expected behavior or if the expected behavior is what is desired.

Since HAZOP is a flow-based analysis, deviations that originate from within components are not inspected directly. Rather, a deviation within a component (as well as a human error or other environmental perturbation) is assumed to be manifested as a disturbed flow [31]. This assumption may be problematic both for manual and automated procedures. A basic strength of the manual HAZOP is the way in which it engenders investigative thought processes. A purely flow-oriented approach may cause the analyst to neglect component-related malfunctions and hazards in favor of pipe-related causes and effects. Thus it may help a manual analysis to hypothesize component deviations.

Software HAZOP

Since a HAZOP concentrates on physical properties of the system [31], it is not directly applicable to analyzing computer input and output. McDermid and Pumfrey [22] outline a manual technique for adapting HAZOP to software design. The procedure taken is essentially identical to a standard HAZOP, except that the pipe-and-process diagram and guide words are changed. The pipe-and-process diagram is replaced by a MASCOT diagram. MASCOT encodes a structural model of the

software design, essentially a data-flow language. The guide words chosen by the authors are listed in table 2.2.

The McDermid and Pumfrey method provides a systematic way to perform a manual HAZOP of software design. The method has been been applied successfully to the analysis of a "moderate size" aerospace system. The authors intend to adapt the method to HOOD and Statecharts.

A problem with developing an automated technique based on MASCOT is that only data flow (or "information flow" in the authors' terminology) is subjected to analysis. While this strategy is faithful to the standard HAZOP procedure, it precludes an analysis based on deviations in system state other than that of data paths. This weakness does not limit the technique's ability to find plausible hazards, since every deviation of a component's state either causes a deviation of an output parameter in the data-flow diagram or else it is not meaningful. Rather, the inclusion of state-based deviations is important because of the exploratory nature of HAZOP. The analysts wish to identify weaknesses in the design, and an analysis that stops at the border of each component does not provide the necessary detail.

Another difficulty in developing an automated technique based on McDermid and Pumfrey's list is the guide word "subtle incorrect". Whereas it is trivial to generate predicates and test cases based on a parameter being "high" (*e.g.*, "$T > T_{max}$" and "$T = T_{max}+1$," respectively), a deviation that is defined to be an erroneous value that "cannot be detected" defies elaboration.

A further difficulty in developing an automated procedure based on MASCOT guide words is the generality of the guide word "COARSE INCORRECT". This single guide word replaces several standard HAZOP guide words, such as "HIGH" and "LOW". Given that a HAZOP analysis is exploratory, the guide words should

Method	Direction	Stages	Automated?
FTA	Backward	SRI	Y
SMHA	Backward	R	Y
FMECA	Forward	S	N
ETA	Forward	S	N
CCA	Combined	S	N
HAZOP	Combined	SD	N

Stages:
S – System requirements
R – Software requirements
D – Design
I – Implementation

Table 2.3: Summary of hazard analysis methods

tend toward specificity in guiding the expert or the computer program. Chapter 5 presents a procedure for developing guide words for specific types of parameters.

2.4 Summary

In summary, there are existing methods for performing automated backward-search hazard analysis of process-control software requirements. In contrast, the analyst does not have a tool to perform a forward search any way but manually (refer to figure 2.3.) The remaining chapters present a theoretical yet practical solution to this problem.

Chapter 3

Deviation Analysis

This chapter presents an introduction to a new hazard analysis method called *deviation analysis.* Deviation analysis uses a forward search to find a causal path from a given initial deviation to a hazardous deviation. Thus, the goal is very similar to that of HAZOP. However, due to the need for the analysis to be at least semi-automated and for the analysis to be applicable to software, the technique is more detailed and must be defined formally. This chapter presents an informal overview of deviation analysis in order to place into context the more rigorous treatment provided in subsequent chapters.

The TCAS II avionics system is used here as an example. The TCAS II system consists of a computer, radio communication equipment, and pilot switches and displays. TCAS II has two parts: a surveillance system and the collision avoidance system (CAS). For each aircraft in the vicinity, CAS determines the aircraft's distance from itself at their closest point of approach (CPA). If the aircraft will be too close, then CAS issues a *resolution advisory,* or RA. An RA is a command to the pilot (via aural alarms and the TCAS display) to take an evasive maneuver, by causing the aircraft to climb, descend, or hold course. Acceptable climb and descend rates are delimited on the TCAS display.

Interface:

Source: Mode-S-Transponder
Destination: CAS
Trigger Event: RECEIVE(Own-Update-Message(Own-Mode-S-ID, Pilot-Selected-SL))
Assignment(s):

$\text{Own-Mode-S-Address}_{\text{v-37}} = \text{Own-Mode-S-ID}$
$\text{Mode-Selector}_{\text{v-34}} = \text{Pilot-Selected-SL}$

Output Action: None.
Description: If an update message is received from own transponder, then update the applicable variables.
MOPS Ref.: Periodic_data_processing (p. 3-P23)

Figure 3.1: Receipt of Mode S Address from the Mode S transponder.

Each operational TCAS II system has a unique identifier called the *Mode S address,* which TCAS II obtains from a radio communication device called the *Mode S transponder.* Figure 3.1 shows the RSML specification of how the Mode S address is received by CAS from the Mode S transponder. The Mode S address and input from the pilot are contained in the *Own-Update-Message.* The values of these fields are assigned to two input variables. (Input variables represent information that has been received from other components.)

One of the key-words of HAZOP is "TOO HIGH." Suppose that the analyst wishes to investigate the potential deviations resulting from field *Own-Mode-S-ID* being too high (*e.g.*, an improperly encoded message.) Inspecting figure 3.1 reveals immediately that TCAS's model of the Mode S address, *Own-Mode-S-Address*, is too high as a result.

The next question the analyst asks is, what system variables does *Own-Mode-S-Address* influence? This variable is used directly in three CAS definitions. Figure **3.2** shows the definition of an output interface in which TCAS II transmits a radio-frequency message, communicating important maneuvering information to a dangerously close TCAS-equipped aircraft. Certain conditions must hold in order for the

high Mode S address to be sent to the other aircraft. The trigger event *Need-To-Send-Resolution-Message* must be present and the guarding condition must be true. Given these circumstances, TCAS II will transmit a resolution message with a high Mode S address for itself. The analyst may then proceed to track the deviation into the receiving aircraft. However, at this point it is clear to a TCAS II expert that the approaching aircraft will not correctly identify the message's origin, thus disrupting safety-critical communication.

A second definition that directly uses *Own-Mode-S-Address* is the macro definition *RA-Display-Delay*, shown in figure **3.3.** This definition addresses the problem of incompatible evasive maneuvers. If two aircraft are equipped with TCAS II units then it is likely that they will both recognize a hazardous situation and attempt to maneuver the aircraft to the same altitude, thus continuing the hazardous situation. The solution is to have one of the units delay issuing an advisory until the other has chosen a direction. Specifically, if the other aircraft also has an operational TCAS II unit *(Other-Capability)* and its maneuvering intent is not known (*Other-VRC*), then the solution is for the TCAS II unit to delay displaying its RA for a certain number of seconds if it has a higher Mode S address.

If the correct value of *Own-Mode-S-Address* is greater than *Other-Mode-S-Address,* then a value that is too high does not change the result. However, if the correct value is less than *Other-Mode-S-Address*, then a too-high value *may* cause *RA-Display-Delay* to be erroneously true. Propagating the deviation thus requires three assumptions:

- The other aircraft has an operational TCAS II unit.
- No intent (RA) has been received from the other aircraft.

Interface:

Source: CAS
Destination: TCAS-Transmitter
Trigger Event: $\text{Need-To-Send-Resolution-Message}_{\text{e-280}}$
Condition:

there exists i :

AND	OR	
$\text{Other-Aircraft}_{\text{s-101}}$ [i] **in state** Waiting-For-Reply	T	·
Some $\text{Other-Aircraft}_{\text{s-101}}$ **in state** Waiting-For-Reply	·	F
$\text{Other-Aircraft}_{\text{s-101}}$ [i] **in state** Waiting-to-Coordinate	·	T

Assignment(s): None
Output Action:

$\text{Sent-RM}_{\text{e-280}}$[i]
SEND(Resolution-Message(Other-Aircraft[i] D $\text{Other-Mode-S-Address}_{\text{v-107}}$,
$\text{Own-Mode-S-Address}_{\text{v-37}}$, Multi-Aircraft-Flag,
$\text{CVC}_{\text{f-232}}$(i), $\text{VRC}_{\text{f-264}}$(i),
$\text{VSB}_{\text{f-265}}$($\text{CVC}_{\text{f-232}}$(i), $\text{VRC}_{\text{f-264}}$(i))))

Description: If the condition is satisfied, then an aircraft model i that satisfies the condition is selected, and a resolution message is sent to the corresponding aircraft. The first column causes i to match with an intruder with which own aircraft has already initiated a coordination sequence. The second column causes i to match with an intruder that has yet to be coordinated with (note that it can only match if own aircraft is not currently coordinating with some other aircraft).
MOPS Ref.: Send_Initial_Intent (p. 6-P57), Complete_Send_Intent (p. 6-P59).

Figure **3.2:** Definition of interface between CAS and the TCAS transmitter for the resolution message command. The table is called an "AND/OR" table. The AND/OR table must be true in order for the assignments to be made. The AND/OR table is true if one of the columns is true. A column is true if each of its rows is true. ('T' = true, 'F' = false, '·' = don't care.)

Macro: RA-Display-Delay

Definition:

A N		
	$\text{Other-Capability}_{\text{v-111}}$ = TCAS-TA/RA	T
	$\text{Other-VRC}_{\text{v-108}}$ = No-Intent	T
	$\text{Own-Mode-S-Address}_{\text{v-37}}$ > $\text{Other-Mode-S-Address}_{\text{v-107}}$	T

Description: Threat is TCAS equipped and no intent has been received and own Mode-S address is higher than threat's.
MOPS Ref.: RESOLUTION_AND_COORDINATION.TCAS_threat_processing.

Figure **3.3:** Macro *RA-Display-Delay.*

- The magnitude of the deviation is greater than the difference in Mode S address values.

Given these assumptions, *RA-Display-Delay* will be true when it should be false.

The macro *RA-Display-Delay* is referenced in five places in the TCAS II specification. All five references are transition definitions. The transitions are shown in figure 2.1, a state diagram representing TCAS's possible classifications of other aircraft. Of interest is the state *Threat,* which is the classification given to an aircraft that appears to be approximately on a collision course with the TCAS unit's own aircraft. Within this state, the *RA-Display-Delay* macro is used to define the transitions of *Advisory,* accounting for three of the five references, and the transitions into *Strength-Not-Selected,* accounting for the remaining two references.

Transition(s): © ⟶ TA/RA-Delay

Location: Threat ᴅ $\text{Advisory}_{\text{s-136}}$
Trigger Event: N/A
Condition:

$\text{RA-Display-Delay}_{\text{m-216}}$	T

Output Action: None
Description: Delay this RA if the other aircraft has a higher Mode S address.
MOPS Ref.: See Intruder-Status macro section.

Figure **3.4:** Transition to *TA/RA-Delay* upon first entering state *Threat.*

Transition(s): Ⓒ ⟶ RA

Location: Threat ▷ $\text{Advisory}_{s\text{-}136}$
Trigger Event: N/A
Condition:

$\text{RA-Display-Delay}_{m\text{-}216}$	F

Output Action: None
Description: Proceed to display RA if own address has precedence.
MOPS Ref.: See Intruder-Status macro section.

Figure 3.5: Transition to *RA* upon first entering state *Threat.*

Advisory indicates whether an RA has been issued against the threatening aircraft. If *Advisory* is in state *TA/RA-Delay* then TCAS is waiting for the threat to communicate its intended RA before taking action. If in state *RA,* then an RA has been issued to the pilot and transmitted to the TCAS-equipped threat.

Figure 3.4 shows the condition for initially entering *TA/RA-Delay.* If *RA-Display-Delay* is erroneously true, then *Advisory* will likewise enter *TA/RA-Delay* erroneously. In order to propagate the deviation, the analyst must assume that *Intruder-Status* was not previously in state *Threat.* This assumption is not inconsistent with the prior assumptions for *RA-Display-Display, so* the deviation is possible.

Figure 3.5 shows the condition for initially entering *RA.* If *RA-Display-Delay* is erroneously true (as we are assuming it is), then the guarding condition for *Advisory* will be false, and the state *RA* will not be entered. Just as with the above transition, the analysis must assume that *Intruder-Status* was not previously in state *Threat* in order to propagate the deviation.

Continuing with the erroneously entered *TA/RA-Delay*: Figure **3.6** shows the definition of the transition from *TA/RA-Delay* to *RA.* If *RA-Display-Delay* is deviantly true, then the first column of the AND/OR table is incorrectly false. However,

Transition(s): TA/RA-Delay ⟶ RA

Location: Threat ▷ $Advisory_{s\text{-}136}$
Trigger Event: $Air\text{-}Status\text{-}Evaluated\text{-}Event_{e\text{-}279}$
Condition:

AND	OR	
$RA\text{-}Display\text{-}Delay_{m\text{-}216}$	F	T
$t \geq t(\text{entered } TA/RA\text{-}Delay_{s\text{-}136}) + 2\ s_{(WTTHR)}$	·	T

Output Action: $End\text{-}RA\text{-}Deferral\text{-}Event_{e\text{-}279}$
Description: Column 1: RA display delay criteria is not satisfied (intent was received from threat). Column 2: RA display delay criteria is satisfied but the delay time limit has expired. (Note: The 2-second delay timeout allows for the reception of two additional surveillance reports. The surveillance update period is nominally one second. The timeout period is subject to change.)
MOPS Ref.: See Intruder-Status macro section.
RESOLUTION_AND_COORDINATION.TCAS_threat_processing, Reversalxheck.

Figure **3.6:** Transition from *TA/RA-Delay* to *RA*.

this deviation is "masked" if at least two seconds have elapsed since entering *TA/RA-Delay*. That is, the transition is taken in the normal way despite *RA-Display-Delay*. The assumption must be made that the elapsed time is less than two seconds in addition to the assumption that the trigger event *Air-Status-Evaluated-Event* occurs in order to propagate the deviation.

The result of the transition being inhibited is that *Threat* is in state *TA/RA-Delay,* it is not in state *RA,* and the output action *End-RA-Deferral-Event* is not produced, all of which are deviations.

The forward analysis should continue in this way for all of the deviations produced. Although the remaining analysis will not be presented in this introductory treatment, the reader will probably not be surprised that the deviations just described, along with further assumptions, cause the suppression of a resolution advisory to the pilot. However, it was not so obvious *a priori* that a spuriously high Mode S address could inhibit the display of an RA.

3.1 Goals

HAZOP has various strengths, and before describing what is expected of the deviation analysis algorithm it would be helpful to eliminate what contribution the algorithm cannot reasonably be expected to make. HAZOP is a group-oriented procedure, by which experts gather in a structured setting to discuss the proposed system. This activity is inherently human in nature and cannot be duplicated by an automated algorithm. Deviation analysis could serve as an additional "expert" during a conventional HAZOP, but the algorithm in this case is simply a resource for the HAZOP procedure.

On the other hand, the role of the HAZOP leader can be duplicated, at least in part. The HAZOP leader is not an expert of the proposed system. This independence from the project helps the HAZOP leader to pose novel questions. Likewise, an algorithm can subject a requirements specification to situations not anticipated by the experts. In particular, a goal of deviation analysis is to capture the utility of the HAZOP guide word for software requirements.

The next chapter presents a primitive language of causality that encodes such information. The analysis also requires the ability to infer how and under what conditions deviations propagate. The calculus of deviations developed in chapter 5 facilitates this task. Finally, chapter 6 presents automated and semi-automated algorithms based on the strategy presented in this chapter.

Chapter 4

A Primitive Language of Causality

This chapter first defines the concepts of "cause" and "system" as used in this thesis. A primitive language of causality is then defined and illustrated based on these definitions. The chapter closes with a description of how RSML specifications may be translated to causality diagrams.

4.1 Definitions

Before investigating the ways in which the causal information may be represented, it may be helpful to define the concept of "cause." Lewycky [19] describes philosophers' attempts through the ages to define cause, and shows how difficult the task is, if it is possible at all. Part of the problem lies in differentiating causation from correlation. If event A always precedes event B, does that mean that A causes B? Fortunately, for the purposes of the hazard analysis algorithm presented in this thesis, the question can remain unanswered. Since all physical relationships are provided by the analyst, the difference between causation and correlation is moot. Both concepts are reduced to logical implication for the purpose of analysis. Thus, if event A is used to define the occurrence of event B, it may be assumed that they are causally linked.

If in fact another event causes both A and B independently, then this does not preclude the analyst from using the occurrence of event A to infer a subsequent B. The only caveat to this practice is that the behavioral definitions must be complete. If C causes A then B, but D causes A only, then A cannot be used to infer the occurrence of B.

Another difficulty is how far back to trace cause. If A causes B which subsequently causes C, then which is the cause of C? Again, this question can remain unanswered for purposes of the algorithm presented in chapter 6. The forward analysis proceeds one step at a time, so cause is always considered to be immediately preceding effect. The analyst can develop more remote definitions of causality from a sequence of events.

It should be noted that the complementary term "dependency" could have been chosen for the discussion, but the concept of dependency is more general than that of causality. In particular, one might confuse use of the term here with the notion as used by dependency graphs in compiler theory. In that sense, a dependency between two variables indicates a relationship of *type*. This concept is similar to, but not the same as, that of value dependencies of a causality diagram. A causality diagram must satisfy certain type constraints, but that is not the purpose of the diagram. For example, given an expression $x = y + z$, where y is an integer and z is a float, a dependency graph reveals that x must be a float. A causality diagram shows that, for example, y's previous value and z's current value are added to produce x's value.

The notion of causality as used in this dissertation can be defined formally. It first requires a formal definition of a system:

Definition 1 (Closed system)

A closed system S is defined by a 5-tuple formal automaton $S = (Q, 6, E, V, B)$, where

- Q is a set of states,
- $\delta : Q \longrightarrow Q$ is the transition function,
- E is a finite set of system variable symbols,
- V is a set of value symbols for the system variables,
- $B : Q \times E \longrightarrow V$ maps system variables to their bound values for a given state.

A closed system's state is defined by the symbols in Q. A closed system has no inputs, but one may think of the "input" to the formal automaton S as an initial state q_0 taken from Q. The transition function δ maps q_0 to the subsequent state q_1, which maps to q_2, *ad infinitum.* A closed system has no outputs, either, but one may consider the series

$$\hat{\delta}(q_0) = q_0\ q_1\ q_2 \cdots$$

to be the "output" of S. If Q is a finite set, then $\hat{\delta}(q_0)$ is a recurring series of length less than or equal to $\|Q\|$.

By definition, a system is a set of interacting components. Each component can be described by one or more *system variables,* such as temperature, altitude, or voltage. Each variable is assigned a symbol in the set E. The function B maps state variables to their values for each state. In practice, a system's state is defined in terms of its state variables, making the two concepts partially redundant. Drawing the distinction aids the following definitions, however.

Recall that the concept of a structural definition was introduced in chapter 2. This notion can be formalized using the above definition of a closed system. The formal definition will be presented in three steps. The first step is to define the concept of a *structural relation.* A structural relation maps the values of a set of variables d to the set of values that a variable r can possibly take. An example should illustrate this definition. Assume a system $S_1 = (Q, \delta, E, V, B)$, where

- $Q = \{q_1, q_2, q_3, q_4, q_5\}$,
- $E = \{A, B, C, D\}$, and
- $V = \{0, 1\}$.

Furthermore, the following table defines B:

Q	A	B	C	D
q_1	0	0	0	1
q_2	0	1	1	0
q	3	0	0	1	1
q_4	0	1	0	1
q_5	1	1	1	0

(The definition of δ is not needed to discuss structural relationships.) Take as an example the structural relation $R_{AB,D}$, mapping the values of variables A and B to D. Referencing the definition of B, the following table describes $R_{AB,D}$:

A	B	$R_{AB,D}(A, B)$
0	0	{1}
0	1	{0, 1}
1	0	{}
1	1	{0}

Starting with the first row, A and B are both 0 in two states: q_1 and q_3. In both of those states, D has a value of 1. The states that apply to the second row are q_2 and q_4. Variable D has different values for those states, hence $R_{AB,D}(0, 1)$ contains two elements. None of the states in Q produces the combination of values for A and B given in the third row, *so* $R_{AB,D}(1, 0)$ is the empty set. Finally, q_5 is the only state that applies to the fourth row, and $B(q_5, D) = 0$.

Since the domain of a structural relation is $2^E \times E$ (a set of variables and a variable), the number of possible structural relations for a system S is the number of sets in the power set of E multiplied by the size of E, or $\|E\| \cdot 2^{\|E\|}$. Applying this formula to system S_1, we see that 64 structural relations are possible.

The concept of a structural relation can be formalized as follows:

Definition 2 (Structural relation)

Given a system $S = (Q,\delta,E,V,B)$, a set of system variables d such that $d = \{d_1, ..., d_n\} \subseteq E$, and a system variable $r \in E$, $R : V_{d_1} \times ... \times V_{d_n} \longrightarrow 2^{V_r}$ describes a *structural relation* between d and r if and only if

$$R(v_1, ..., v_n) = v_r \equiv$$
$$\forall q \in Q : (B(q,d_1) = v_1 \wedge ... \wedge B(q,d_n) = v_n) \Rightarrow B(q,r) \in v_r$$
$$\wedge\ \forall i \in v_r : \exists q \in Q : (B(q,d_1) = v_1 \wedge ... \wedge B(q,d_n) = v_n)$$

The definition is a bi-implication in the form of two constraints. Put simply, the definition asserts that a value i is in v_r if and only if there is a state with the appropriate values for the state variables in d. The first constraint corresponds to the "if" part of the bi-implication: if the variables in d have the values $v_1, ..., v_n$, then the value for r must be in v_r. The second constraint (the "only if" part) asserts that for each value i in the set v_r, there must be a state q that has the variables in d bound to the values $v_1, ..., v_n$ and variable r bound to i.

To relate this definition to the example for $R_{AB,D}$ above, d is the set $\{A,B\}$ and r is the variable D. In the course of defining R, d_1 and d_2 are arbitrarily chosen to refer to A and B, respectively. Thus, if $v_1 = 0$ and $v_2 = 1$, then inspection of $R_{AB,D}$'s table shows that $v_r = \{0,1\}$.

Note that there exists exactly one structural relation between any set of variables d and variable r. The existence of two relations R_i and R_j implies that there exists a set of bindings for d such that the two relations differ. Since they differ, the value for one of the relations, say R_i, must contain an element v_i that R_j does not contain. If indeed r may take the value v_i given the bindings for d, then R_j fails the first clause of the above definition and is not a relation for d and r. If r does not take

the value v_i for any state with the given bindings, then R_i fails the second clause and is not a relation between d and r.

The range of a structural relation is a set of variable values. If the range of the relation contains exactly one value for each mapping from d to r, then the relation is a function. This fact is very helpful in using the variables in d to determine the value of r. Formally, a structural function is defined as follows.

Definition 3 (Structural function)

Given a system $S = (Q, \delta, E, V, B)$, a set of system variables $d \subset E$, and a system variable $r \in E$, a *structural function* exists between the bindings of d and r if and only if

$$\text{functional}_0(d, r) \equiv$$
$$\forall i, j \in Q : (B(i,r) \neq B(j,r)) \Rightarrow (\exists k \in d : B(i,k) \neq B(j,k))$$

In other words, if variable r has different values for two states i and j, then some variable in d must also have different values for i and j. Otherwise, r has multiple values for some single set of values for d and the relation that maps the values of d to r is not a function.

Structural functions have a property that is useful to the definition of causality as used in this thesis. If a structural function exists mapping values for a set of system variables d to a single value for another variable r, then the variables in d can be used to infer the value of r. Strictly speaking, r is functionally redundant with respect to the other system variables. The existence of functionally redundant system variables is not an indication of a poorly written specification. Rather, redundancy is commonplace in the description of physical properties of the system and the controller's model

of these properties. For example, the ideal gas law, $PV = nRT$, does not obviate the explicit reference to a reactor's pressure, volume, mass, and temperature in a specification.

Thus, structural functions indicate the presence of structure in the specification — some variables are defined in terms of others. However, the definition is somewhat weak in that it shows that a set of variables is *sufficient* to define another variable, but not that all the members of that set are *necessary* for the definition. For example, mapping all the variables in the system to a single variable r is certainly a structural function, but it contains variables that are not structurally related to r. The following definition of *structural causality* includes in d only those variables that are necessary to describe r unambiguously, *i.e.,* necessary as well as sufficient:

Definition 4 (Structural causality) ________________________________

Given a system $S = (Q, 6, E, V, B)$, a set of system variables d C S, and a system variable $r, r \notin d$, d *structurally causes* r if and only if

$$\text{functional}_0(d, r) \;\text{A}\; \forall i \subset d : \neg\text{functional}_0(i, r)$$

This definition states that a functional mapping from d to r must exist, and there must not exist a functional mapping from a subset of d to r. The existence of such a mapping would indicate the existence of an independent variable in d that is independent of r.

Sequential causality can be defined in a way analogous to structural causality:

Definition 5 (Sequential function, sequential causality) ____________________

1. Given a system $S = (Q, 6, E, V, B)$, a set of system variables d C S, and a system variable r, a *sequential function* exists between the bindings of d and r

if and only if

$$\text{functional}_1(d,r) \equiv$$

$$\forall i,j \in Q : (B(D(i),r) \neq B(D(j),r)) \Rightarrow (\exists k \in d : B(i,k) \neq B(j,k))$$

2. Further, d *sequentially causes* r if and only if

$$\text{functional}_1(d,r) \wedge \forall i \in d : \neg\text{functional}_1(i,r)$$

Finally, the two concepts can be combined for a general concept of causality:

Definition 6 (Functional relationship between variables)

1. Given a system $S = \langle Q, \delta, E, V, B \rangle$, two sets of system variables $d_0, d_1 \subset S$ (not necessarily disjoint), and a system variable r, a *function* exists between the previous bindings of d_1, the current bindings of d_0, and the current bindings of r if and only if

$$\text{functional}(d_1, d_0, r) \equiv$$

$$\forall i,j \in Q : (B(D(i),r) \neq B(D(j),r)) \Rightarrow$$

$$(\exists k \in d_1 : B(i,k) \neq B(j,k)) \vee$$

$$(\exists k \in d_0 : B(D(i),k) \neq B(D(j),k))$$

2. d_0 and d_1 *cause* r if and only if

$$\text{functional}(d_1, d_0, r) \wedge \forall i \in d_1 : \forall j \in d_0 : \neg\text{functional}(i,j,r)$$

4.2 Encoding Causality

Although the deviation analysis algorithm can be developed for each separate specification language, there are considerable advantages in developing the algorithm and calculus around a "core" analysis language. The language should have a basic syntax and semantics into which requirements languages can be translated. There are several reasons for doing this. Requirements languages often have subtle semantics. It would aid development of the analysis algorithm if the semantics relevant to analysis can be made explicit. It is also an advantage to divide the research problem into that of (1) translating the relevant parts of the language into a more fundamental representation and (2) developing an analysis algorithm for the simpler language. In addition, this tactic aids adaptation of the algorithm to other requirements languages, since it is an easier task to translate from a full requirements language to the simpler analysis language than to translate to another full-featured requirements language or to rewrite the analysis algorithm, since it very likely makes semantic assumptions. Finally, it is more difficult to explain and understand the algorithm if it is defined over a language with features irrelevant to deviation analysis. Thus, it is preferable to develop and present the algorithm using a basic analysis language.

A potential disadvantage of using a separate analysis language is that the analyst must maintain two mental models: one of the specification and one of the analysis model. However, the translation from specification to analysis language is automatic, so the analyst does not need to see the analysis model for the purpose of input to the algorithm. In addition, an entity in the analysis model always maps back to a single entity in the specification. Thus the algorithm's results (including a full search tree) can be satisfactorily presented to the analyst in terms of the specification model.

Therefore it is not strictly necessary that the analyst inspect the analysis model at all.

What is desirable in a language of causality? First of all, the relationships represented should be direct. That is, there should be no intermediate variables or "steps" that take no time. The language must be able to represent complex relationships in order to propagate deviations. The notion of sequential dependency should also be explicit, rather than relying on relationships implied by the language's semantics. Simultaneous relationships should be explicit. The language must be able to handle numeric and boolean system variables.

Several languages already exist that partially encode causality. However, they do not encode enough information about the nature of causal relationships to be useful for a software hazard analysis.

A *dependency graph* is a type of directed graph that indicates the "interdependencies among the inherited and synthesized attributes at the nodes in a parse tree." [1] The relationships are restricted to type, and thus are basically limited to analysis of structural causality.

In contrast, pipe-and-process and block diagrams show sequential causality but generally lack structural causality. Additionally, the nature of the sequential causality is not described, so only the most rudimentary hazard analysis may be performed automatically.

Digraphs (section 2.1.2) encode sequential causality as a partial derivative. However, with the exception of the Logic Flowgraph Method (LFM) [9], interaction between causes is not represented. Even with the limited specification of interactions

allowed by LFM, the partial derivative is inadequate to specify much of the control behavior of software.

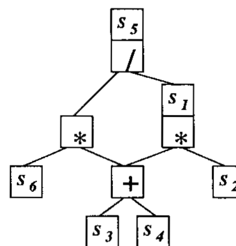
4.3 Causality Diagram Grammar

A causality diagram is composed of a set of nodes N, connected by directed edges. Each node $n \in N$ has a function associated with it that, combined with the edges into n, defines its causality. The value of the nodes is the state of the causality diagram.

Nodes in a causality diagram are divided into *source nodes* and *auxiliary nodes.* The source nodes correspond to the system variables. The causality of a source node is often a complex expression; the auxiliary nodes serve to compose functions of source nodes. For example, assume the following arithmetic causalities between source nodes s_1, s_2, s_3, s_4, and s_5,

$$
\begin{aligned}
s_1 &= s_2(s_3 + s_4) \\
s_5 &= \frac{s_6(s_3 + s_4)}{s_1}
\end{aligned}
$$

These equations can be represented by the following tree:



Each of the unnamed nodes represents an "intermediate" expression, in this case arising from operator precedence. The expressions have value, and the deviation analysis algorithm exploits this fact to provide some information when a deviation

cannot propagate from one source node to another. For example, suppose that s_3 is less then it should be but s_4 is normal. Then the sum of s_3 and s_4 is less than it should be. However, if the values of s_2 or s_6 are not known then the deviation cannot propagate to another source node without making any assumptions. It is useful to know though that those expressions are deviant.

Furthermore, the calculus used by the search algorithm to propagate the deviations may be defined for functions of fixed arity. For example, the sum $a+b+c+d$ may be parsed as $a+(b+(c+d))$ for purposes of analysis. Such a restriction requires additional nodes (one for each set of parentheses.)

Finally, the model specified by the requirements may be incomplete, in essence making it an open system with unspecified sources and sinks. In consideration of this and the aforementioned needs, the language of causality diagrams contains intermediate nodes (called *auxiliary nodes),* which are considered to be the same as source nodes for the purposes of propagating deviations. For example, the following auxiliary nodes may be defined as

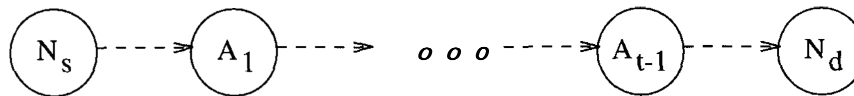
$$\begin{aligned} a_1 &= s_3 + s_4 \\ a_2 &= s_6 + a_1 \end{aligned}$$

such that s_1 and s_5 above may be defined by binary arithmetic operators and auxiliary nodes:

$$\begin{aligned} s_1 &= \text{plus}(s_2, a_1) \\ s_2 &= \text{divide}(a_2, s_5) \\ a_1 &= \text{times}(s_3, s_4) \\ a_2 &= \text{plus}(s_1, a_1) \end{aligned}$$

An edge is described for each parameter in a node N_d's function domain. Each edge may be represented as a 2-tuple $\langle N_s, T \rangle$, where N_s is the source node and $T \in$ *{sequential,structural}*. The edge associates the range of N_s's function with part of N_d's domain. If $T =$ *sequential,* then the causal relationship between N_s and N_d is temporal. The value of N_s in one step affects the value of N_d in the next step. If $T =$ *structural* then the causal relationship is structural, which is to say that the current value of N_s constrains what the current value of N_d is. The terminology used in this thesis is that the source of an edge (N_s) is called the *parent* node and the node at the terminus (N_d) is called the *child* node.

Because auxiliary nodes facilitate functional composition, the edges with an auxiliary node as source almost always have $T =$ *structural.* The exception is when an auxiliary node serves to describe a sequential relationship that extends beyond one step, e.g., if node N_s influences node N_d's value with a time lag of t steps, then $t - 1$ auxiliary nodes are needed:



(Sequential edges will be represented by dashed arrows in this thesis. Structural edges will be represented by solid arrows.)

A node may have a sequential relationship with itself, *i.e.*, it may be the source of an edge with $T =$ *sequential.* However, there may not be any loops along paths composed only of structural edges. This situation is easy to discover by a search of the directed graph created by the nodes and structural edges.

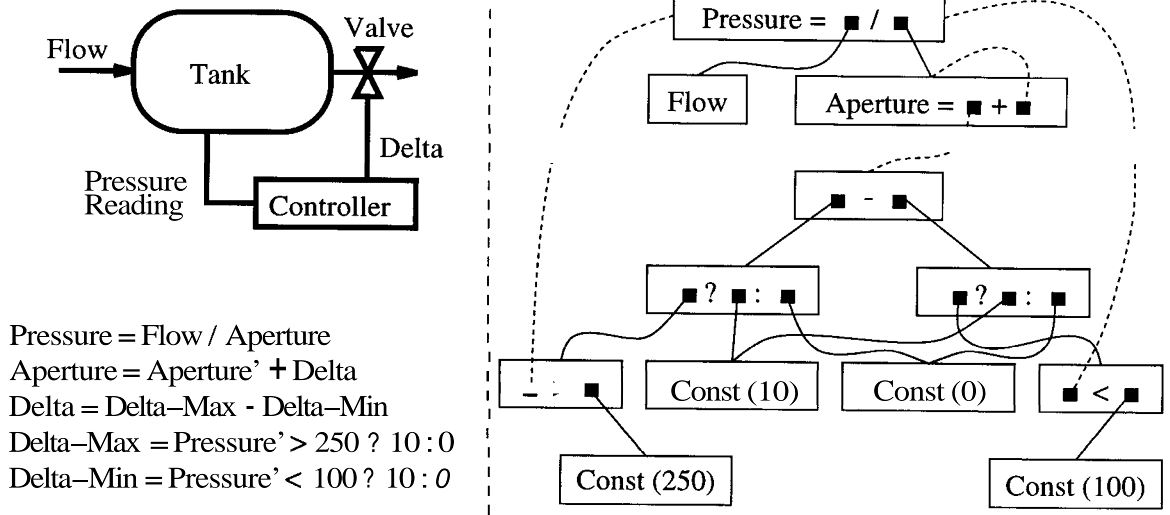


Figure 4.1: Causality diagram example

Putting the concepts of structural and sequential causality together, figure 4.1 shows an example of a simple feedback system and the corresponding causality diagram. The system shown is a tank equipped with a variable-aperture valve. The system variables are the tank pressure, the flow of material through the tank, and the aperture of the valve. To simplify the example, pressure is the quotient of flow over aperture. The controller increases or decreases the valve opening by ten units if the pressure is above the maximum of 250 units or below the minimum of 100 units, respectively.

The causality diagram in the example contains twelve nodes and sixteen edges. Three of the nodes represent the system variables. The behavior of the *Flow* variable is undefined. *Pressure* is a quotient function, with the numerator edge originating from *Flow* and the denominator edge originating from *Aperture*. *Aperture* is an interesting node because it has direct feedback from its previous value. The size of the valve aperture is equal to its previous value (indicated by a dashed line) plus one of $\{-10, 0, 10\}$, as provided by the controller.

The remaining nodes comprise the controller. The pressure reading is compared to minimum and maximum values (the “$<$” and “$>$” nodes, respectively.) Note that each node is a function: The domain of the inequality functions is a pair of numbers and the range is a boolean.

The nodes represented by $\boxed{\square \; ? \; \square \; : \; \square}$ are called *selection nodes.* The selection function is defined as follows:

$$b \; ? \; x : y = \begin{cases} x & \text{if } b \\ y & \text{if } \neg b \end{cases}$$

Thus, following the edges from the subtraction node (the output of the controller) backward to the pressure reading, one gets the expression

$$[(Pressure' > 250) \; ? \; 10 : 0] - [(Pressure' < 100) \; ? \; 10 : 0]$$

where *Pressure'* represents the previous value for pressure. This is the value that is output to the valve actuator.

4.4 Translation of RSML

This section describes how to translate the Requirements State Machine Language (RSML) to the primitive language of causality. RSML is a graphical, state-based requirements specification. RSML specifications are composed of interconnected system components. Each component is composed of a heirarchical state definition, a useful state abstraction originally developed for the Statecharts specification language [10]. Please refer to [11] for a presentation of RSML. Some review of RSML accompanies the following discussion of translation issues, but knowledge of the language's semantics is recommended.

$\mathcal{B} \rightarrow \mathcal{B}$	identity, negation
$\mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$	and, or
$\Re \rightarrow \Re$	identity, negation
$\Re \times \Re \rightarrow \Re$	$+, -, \times, \div$
$\Re \times \Re \rightarrow \mathcal{B}$	$=, \neq, <, >, \leq, \geq$
$\mathcal{B} \times \Re \times \Re \rightarrow \Re$	selection
& + &	identity
$\mathcal{E} \times \mathcal{E} \rightarrow \mathcal{B}$	$=, \neq, <, >,$ [illegible] $\geq$
$\mathcal{B} \times \mathcal{E} \times \mathcal{E} \rightarrow \mathcal{E}$	selection

Table 4.1: Basic functions. "$\mathcal{B}$" is the set of booleans, "$\Re$" is the set of reals, and "$\mathcal{E}$" is an enumerated set.

4.4.1 Basic Functions

Each node in a causality diagram is associated with a particular function. This section presents the basic functions necessary to translate RSML. Although most of the functions given here should be useful for translating another language to causality diagrams, it is likely that additional basic functions would need to be defined. For example, RSML does not include any set theoretic functions.

The functions described here can be divided into three groups: functions over the booleans, reals, and enumerated types. These functions are listed in table 4.1. Most of the functions are self-explanatory. The "selection" function has already been defined. Enumerated types are small sequences $\{e_1, e_2, ..., e,\}$. The relational operators are defined such that

$$e; \ = \ e_i$$

$$e_i < e_j \ \Leftrightarrow \ i < j$$

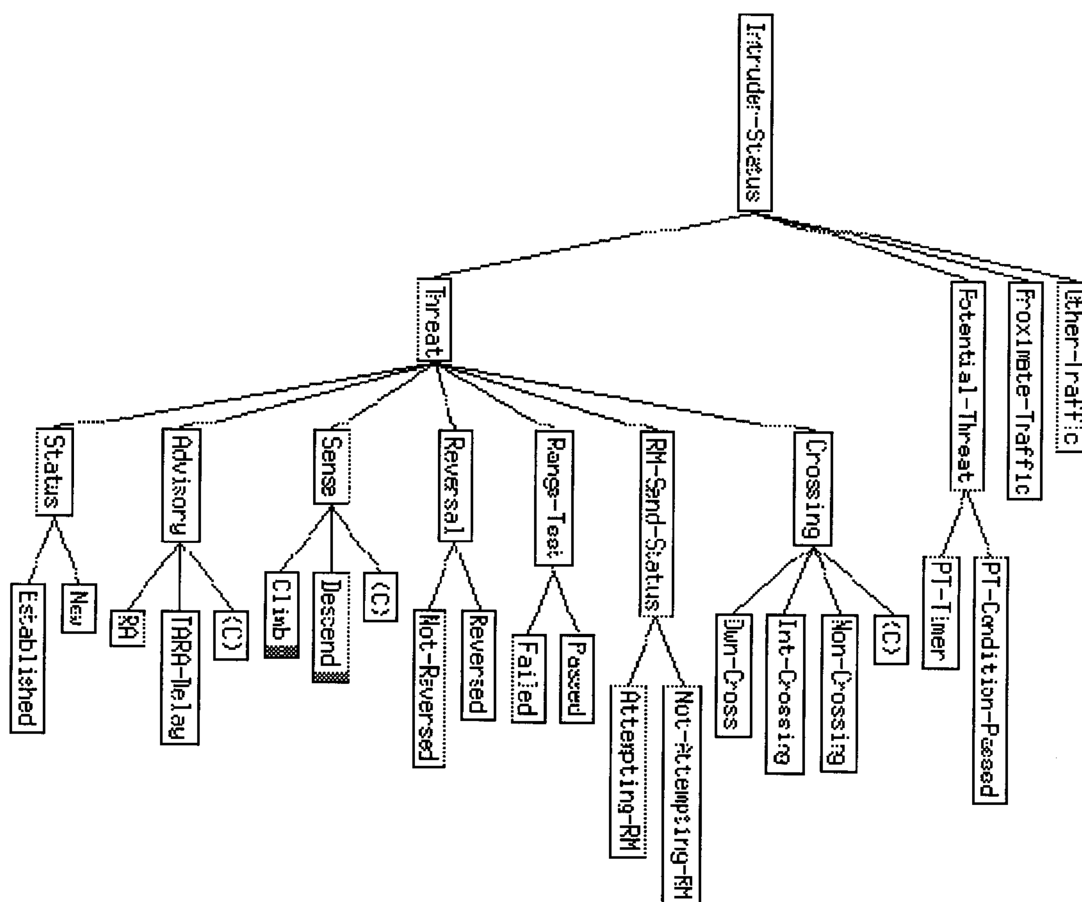


Figure 4.2: The *Intruder-Status* state heirarchy.

4.4.2 States

An RSML specification is composed of a heirarchy of "states." For example, figure 4.2 shows the state heirarchy for *Intruder-Status* (figure 2.1.) According to the definition of a system given earlier in this chapter, RSML states are actually state variables because they represent part of the system state. The translation of an RSML specification to a system automaton would involve mapping the RSML state symbols to the variable symbols V. Since the terminology is somewhat conflicting, please interpret use of the term "state" to be an RSML state variable for the remainder of this section.

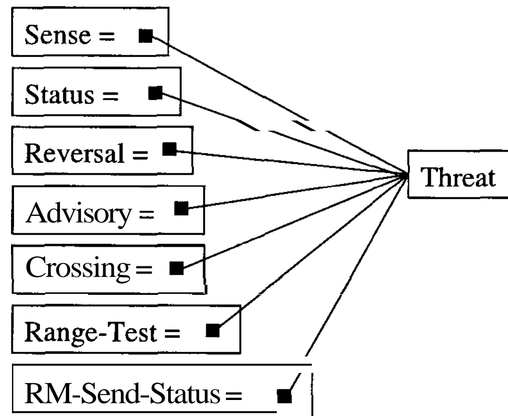


Figure **4.3:** The portion of the causality diagram showing the relationship between *Threat* and its children. Each child takes its value directly from *Threat.*

States are binary variables; they are either "active" or "inactive." These two values can be modeled by boolean nodes in the causality diagram (*i.e.*, functions with a boolean range.) States are classified by their children. **A** state with no children is called a *basic state.* Two basic states in figure 2.1 are *Other-Traffic* and *New.* **A** state whose children are mutually-exclusive is called an *or-state.* If the or-state is active, then exactly one of its children is active. *Intruder-Status* and *Crossing* are examples of or-states in figure 2.1. The children of an *and-state* are all active if the and-state is active. *Threat* is an example of an and-state.

And-States

The causal relationship between an and-state and its children can be described by the boolean identity function. That is, the value of the children (true or false, active or inactive) is exactly the same as the and-state's value. Figure **4.3** shows a partial causality diagram for state *Threat.*

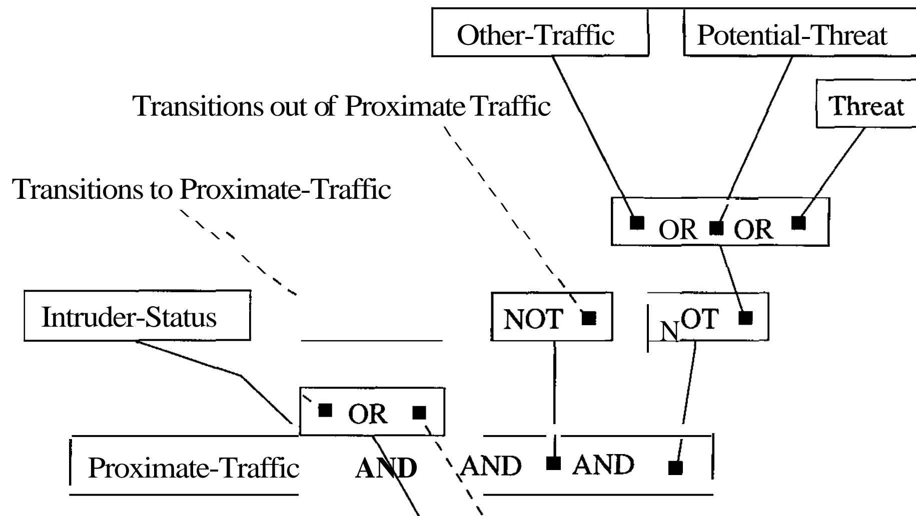


Figure 4.4: A causality diagram fragment for *Proximate-Traffic*.

Or-States

The causal relationship between an or-state and its children is a bit more complicated. The or-state must be active in order for any child to be active. Three additional conditions must hold in order for a child state to be active:

- There was either a transition to the state or the state was active in the previous instant. The latter condition means that once in a state the system stays in that state until there is a transition to another state. This characteristic contrasts with transient variables, such as events.
- There was not a transition out of the state in the previous instant.
- None of the siblings of the state are active.

he first two conditions are sufficient to describe the behavior of a state procedurally. However, the third condition is needed in order to enforce the mutual exclusion of or-states. For example, if the analyst postulates an initial scenario in which *Climb* and Descend are both active, the first two conditions would not form a contradiction.

The third condition would cause the two nodes to be true and false simultaneously, signaling an impossible scenario.

Figure 4.4 shows a causality diagram fragment based on the relationship between *Proximate-Traffic* and its parent and siblings.

4.4.3 Events

Events may be thought of as transient states. In RSML and similar languages, events are a sort of communication between states. Events are also used to synchronize state transitions. In this capacity they may be thought of as similar to the parentheses of an equation [11].

Events can be produced by state transitions, input interfaces, output variable transitions, and output interfaces — all of which are types of component transition. Since the same event can be produced by multiple transitions, an event is simply an *or* function of the various transitions that can produce it. In RSML, events are considered to be active in the next instant after the transition is taken, so all of the edges into the *or* function are sequential.

The next section shows an example of a transition producing an output action.

4.4.4 Transitions

Transitions have five components:

- source state,
- destination state,

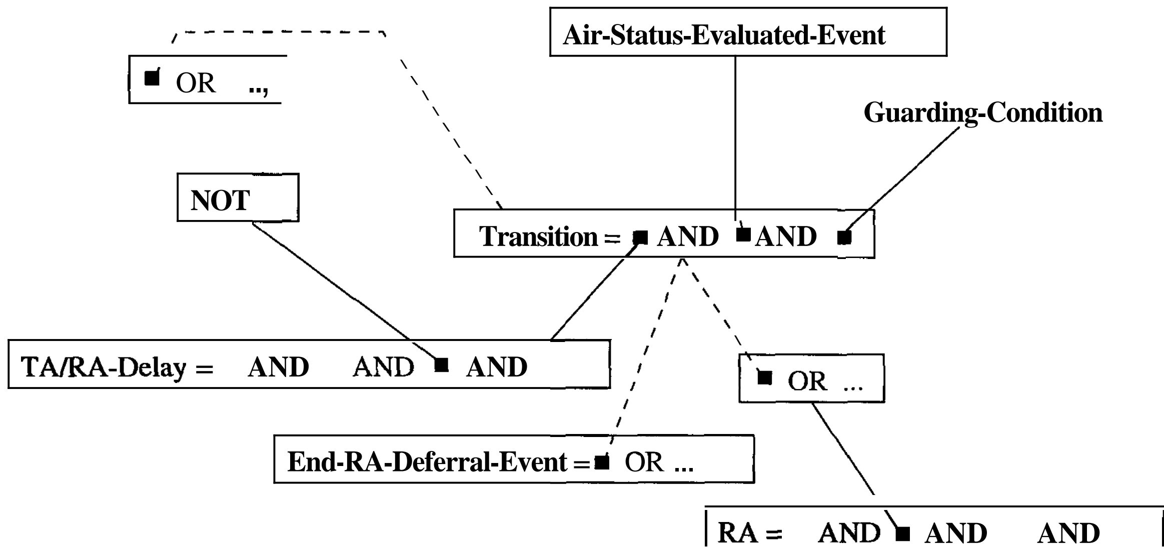


Figure 4.5: A causality diagram fragment for the transition from *TA/RA-Delay* to *RA*. Compare this diagram to the or-state example to see how transition definitions combine with or-state definitions.

- triggering event,
- guarding condition (optional), and
- output action (an optional event).

The semantics of a transition may be described by the following logical inference:

$$S \wedge E \wedge C \Rightarrow \neg S' \wedge D' \wedge A',$$

where S, E, and C are the values of the source state, triggering event, and guarding condition in one instant, and S', D', and A' are the values of the source state, destination state, and output action in the next instant. A causality diagram fragment for the transition in figure 3.6 is shown in figure 4.5.

Conditional Connectives

Conditional connectives (represented by '©') are an abstraction for combining transitions. Transitions out of conditional connectives have two distinct differences

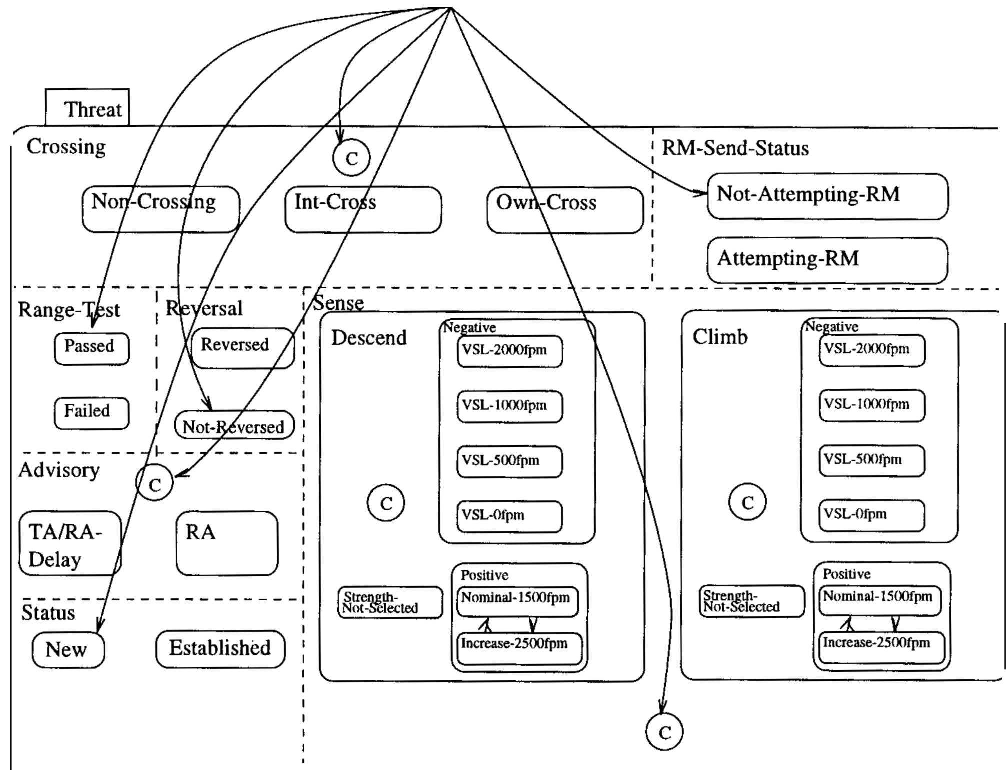


Figure 4.6: **A** transition into *Threat* is actually a transition into all of the default states contained in *Threat.*

from normal transitions: (1) trigger events are not allowed, and (2) the transition from the conditional connective to its destination takes no time. Thus, the causality diagram for a transition out of a conditional connective differs from a normal transition in that a triggering event is not an input to the transition's AND function and the edge from the transition to the destination state is coexistent rather than sequential.

Transitions into conditional connectives are the same as any other transition.

Default State Transitions

Or-states may have *default* states. The simplest method of translating default transitions to a causality diagram is to translate it first to its component transitions

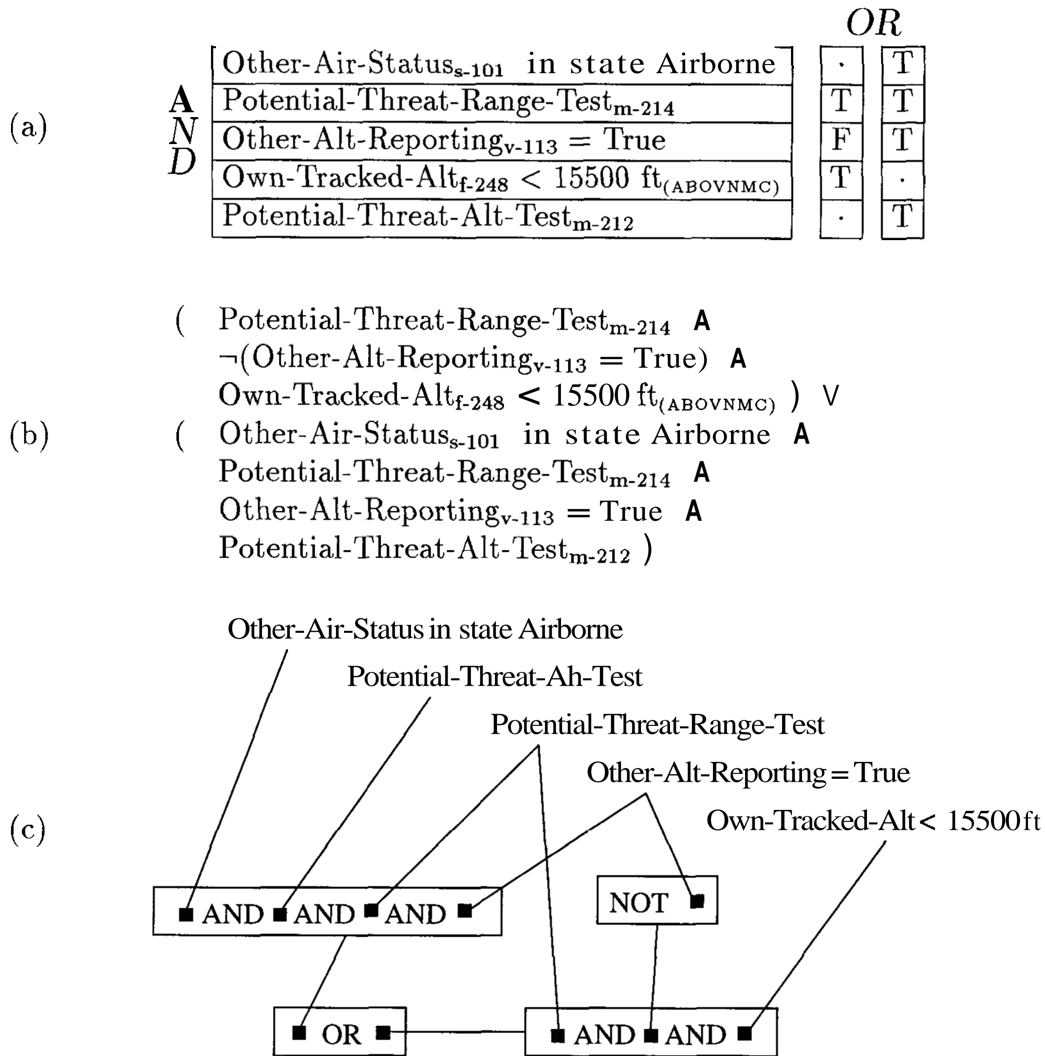


Figure 4.7: AND/OR Table.

(see figure 4.6.) Such a construction would be very difficult to read, but it makes the causal relationships quite clear.

Transition Bus

The transition bus is simply a graphical shorthand for transitions directly between states; its purpose is to minimize the number of arrows on the page and show

the interconnectedness of states. Hence it does not contribute to the semantics of RSML or to causality diagrams.

4.4.5 AND/OR Tables

AND/OR tables are a graphical representation of boolean expressions in disjunctive normal form. As such, the semantics of AND/OR tables are quite straightforward and easy to encode in a causality diagram. The left-most column of each AND/OR table lists the terms of the expression. The remaining columns are logically OR'd, so that the table is true if only one column is true. A column is true if all of its terms are true. For example, figure 4.7(a) shows an AND/OR table from the TCAS II specification. Figure 4.7(b) shows the same expression in conventional boolean notation. Figure 4.7(c) is the same table as a causality diagram fragment.

4.4.6 Macros

A macro is an abstraction of an AND/OR table. RSML has two kinds of macro: parameterized and non-parameterized. Non-parameterized macros are simply AND/OR tables with names and the translation proceeds in exactly the same way.

Parameterized macros have one or more parameters in their AND/OR tables. A reference to a parameterized macro replaces each parameter with an expression, such as the name of an input variable or a state.

Since causality diagrams represent relationships directly, each parameterized macro reference must be replaced by the causality diagram fragment of the AND/OR table, with the parameters instantiated.

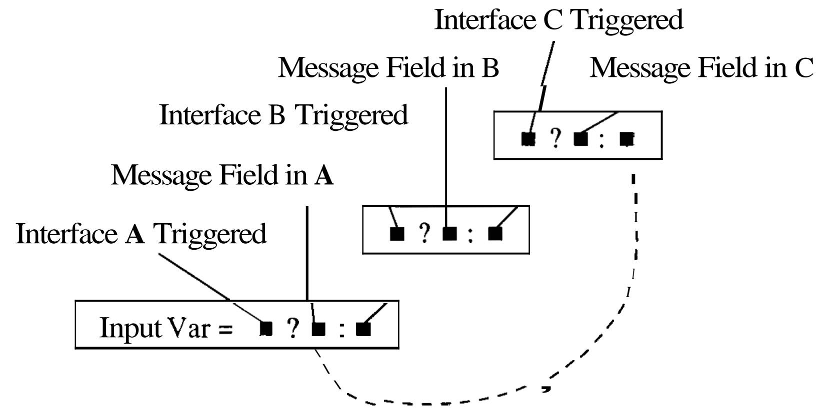


Figure 4.8: Causality diagram template for input variables

4.4.7 Functions

Like with parameterized macros, each function reference must be replaced by a causality diagram of the function definition with the function parameters instantiated.

4.4.8 Input Variables

Input variables are set by input interfaces. They may be represented as a series of selection nodes, as shown in figure 4.8. In the example, the input variable can be assigned a value by one of three interfaces A, B, and C. If none of the three interfaces is triggered by the receipt of a message, then the input variable retains the value it had in the previous instant.

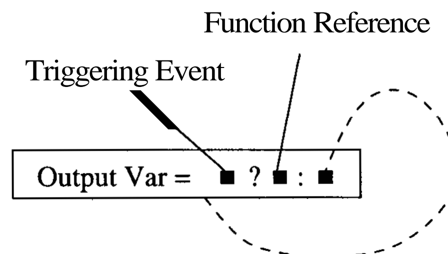


Figure 4.9: Causality diagram template for output variables

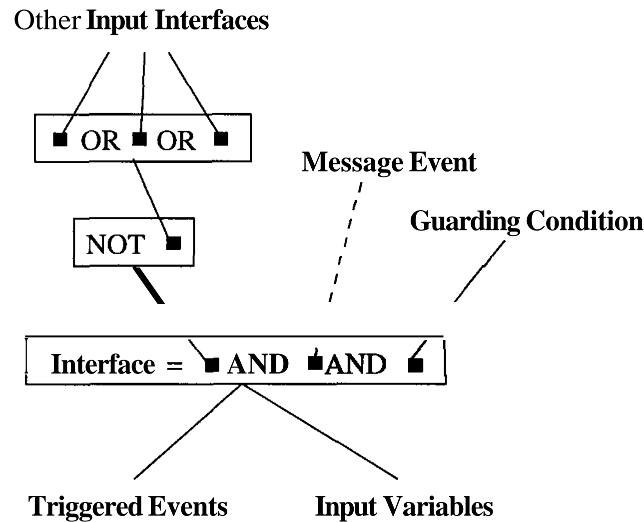


Figure 4.10: Causality diagram template for input interfaces

4.4.9 Output Variables

Changes in output variable values are triggered by events. The new value is given by the value of a function reference at the time of the event. Figure 4.9 shows a causality diagram fragment.

4.4.10 Input Interfaces

Input interfaces receive messages from other components. The message has two parts: a boolean node representing whether the message has been sent and a numeric (or enumerated type) node for each of the fields of the message. The interface is active if the message is active and the input interface's guarding condition is true. Input interfaces can can events to be active. Input interfaces share these characteristics with transitions.

Like or-nodes, input interfaces are mutually exclusive, so a condition of an interface being active is that the others are inactive. Figure 4.10 shows a causality diagram template for input interfaces.

4.4.11 Output Interfaces

Output interfaces package output variables into a message and send to a component or components. The output interface is triggered by an event, contingent on a guarding condition. In addition to the message it may also generate events within the scope of its own component.

4.5 Summary

This chapter began with a definition of closed systems, and especially system variables. This definition was used to define two types of causality as related to requirements specifications, The concepts of causality were incorporated into a primitive language of nodes and edges, the artifacts of which are termed causality diagrams. Finally, a strategy for translating RSML specifications to causality diagrams was presented.

Chapter 5

A Calculus of Deviations

The previous chapter introduced a primitive language of causality, by which system variables are defined in terms of present or previous values of other system variables. One possible use of the causality diagram is testing. If certain nodes are given specific values then the simultaneous values of other nodes could be derived based on structural relationships. Likewise, subsequent values of nodes could be determined by inspecting sequential relationships. This would be similar to a partial "execution" of the specification. This information would be useful, but the results would be valid only for the tests performed. Consequently, the analyst must rely on proper coverage of the state space just as in execution of the specification in the source language. The causality diagram may still have advantages for this purpose. For example, only relevent parts of the specification are "executed." Furthermore, the causality diagram may be an appropriate language upon which to develop test selection methods based on deviations in the environment (as opposed to testing under normal operating conditions.)

Alternatively, the causality diagram could be used to develop a proof of the system's behavior. Although formal verification of safety constraints is more general than testing, it is also more difficult. For many systems it is just not practical to develop and review a proof.

These two common approaches are on a continuum of feasibility versus generality of the analysis. In practice, compromises must be made between the costs of constructing and analyzing the system model and the quality of the results [16].

The particular balance struck depends in part on the stage of development. The system model typically begins as a coarse, incomplete description of the problem and is amended and refined as the solution develops. Accordingly, analysis should be an iterative process [16]. To wit, preliminary analysis should readily provide coarse results for incomplete information and final testing should yield conclusive results for an operational system.

The general topic of this thesis, software requirements analysis, is one of the earliest stages of software development (though not quite as early with respect to the system as a whole.) At this stage, the information is usually incomplete and volatile. Accordingly, the analyst needs procedures that can expeditiously furnish results given incomplete, abstract information. On the other hand, as the control model is being developed, the analyst does not require conclusive results so much as motivating information.

5.1 Introduction To Qualitative Mathematics

A potential solution lies with *qualitative mathematics.* Qualitative mathematics is the creation and study of calculi of small ordered sets, called *qualitative domains.* Qualitative domains partition the system's *quantitative domains* (usually the set of real numbers.) Formally, a qualitative domain is defined by a function mapping members of the quantitative domain to members of a small set. In other words, if $\mathcal{D}$

$x + y$

	−	0	+	?
−	−	−	?	?
0	−	0	+	?
+	?	+	+	?
?	?	?	?	?

$x \times y$

	−	0	+	?
−	+	0	−	?
0	0	0	0	0
+	−	0	+	?
?	?	0	?	?

Table 5.1: Sign algebra.

is some domain (*e.g.*, the integers or complex numbers) and $\mathcal{L}$ is a (small) finite set, then a function $M : \mathcal{D} \longrightarrow \mathcal{L}$ defines $\mathcal{L}$ as a qualitative domain over $\mathcal{D}$.

The area of research that has introduced qualitative mathematics is most commonly referred to as "qualitative reasoning," although variations in research emphasis have led to such labels as "causal reasoning," "qualitative process theory," and "qualitative analysis." Qualitative reasoning has been proposed as a method for system control, as an educational tool, and for system analysis. The present thesis falls into the latter category and so can be termed more accurately as qualitative analysis.

Note that when viewed from the perspective of a calculus, the causality diagram may be seen to be a set of axioms. Each node N of the diagram can be rewritten as $N = f(I_1, \ldots)$, where f is the node's function and $I_1, \ldots$ are input nodes. An analysis procedure may apply the calculus to the set of system axioms to produce "theories" of system behavior.

A simple and commonly used qualitative domain is the set of signs of the real numbers, $S_Q = \{-, 0, +, ?\}$. S_Q partitions the real numbers into two sets, the positive (+) and negative (−) numbers. Zero (0) is the border between the two sets. The special symbol "?" represents an unknown value and is equivalent to the union of the other symbols. The addition and multiplication functions over S_Q are referred to as *sign algebra* in the literature. Their definitions are shown in table 5.1.

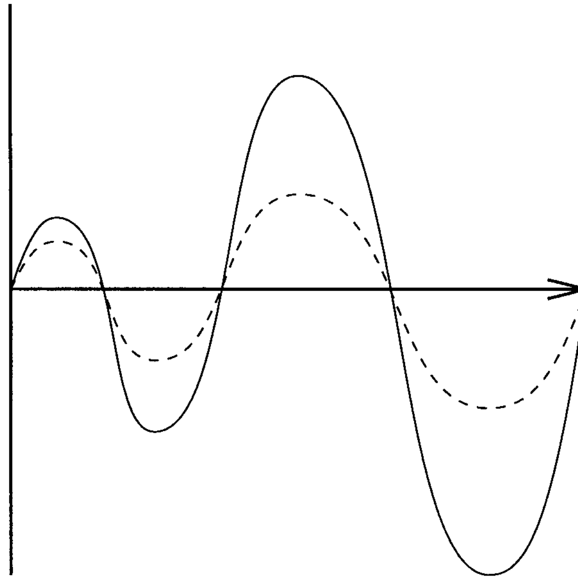


Figure 5.1: A qualitative representation of oscillating functions. The dashed line is an oscillating function with increasing envelope, descreasing frequency, and normal shape. The solid line is the same function with "sharp" shape.

Qualitative mathematics is not limited to algebraic functions. For example, Schaefer constructs an interesting model for a family of nonlinear oscillating functions [29]. The parameters of the functions are the envelope, period of oscillation, and "shape" of the function. Figure 5.1 shows an example of a qualitative oscillating function. Schaeffer has developed an algorithm for qualitatively solving the derivatives of this family of functions. Please see [29] for a detailed treatment of this calculus.

Qualitative mathematics has the advantage of being efficient to calculate and relatively easy to understand. Similar values can be grouped and treated collectively by the qualitative functions.

A disadvantage of qualitative analysis is that potentially useful information is lost in the discretization of the quantitative domains. This is not unusual, as all models are incomplete approximations of reality. The analyst must decide whether

a qualitative analysis, or any method of analysis, is appropriate and useful for a particular system.

It should be noted that the logic used in digraph models such as LFM is a type of qualitative mathematics. In particular, they represent qualitative partial derivatives of process variables. The qualitative domain is composed of the set of values $\{-10, -1, 0, +1, +10\}$. The digraph algebra is composed of such rules as $(-1) + (\$10) = +10$ (a small negative influence combined with a large positive influence results in a large positive influence.)

A problem with the digraph algebra is that while the qualitative calculations are internally consistent, the calculus is not consistent with respect to the quantitative domain. For example, suppose that $L > 2$ is the boundary between small and large values. $(L - 1)$ and 2 are both small positives, and the sum of two small positives is a small positive under digraph algebra. But $(L - 1) + 2 = L + 1$ is large, so the result of adding two numbers and converting the result to a digraph value is inconsistent with converting the two numbers and then adding them qualitatively.

5.2 $P_{B,N}$: A Logarithmic Qualitative Domain

Before describing the calculus of deviations it is first necessary to describe the qualitative calculus for the value part. Since booleans and enumerated types are small sets, they can be analyzed directly without converting to qualitative sets. In fact, they may be considered to be qualitative sets since they partition the system state space into a small number of salient values.

The author has chosen a logarithmic mapping from the numbers to a qualitative domain. A logarithmic scale allows coverage of a large range of values while still partitioning the smaller values. The following function defines a mapping from the real numbers to a family of qualitative domains. The parameters of the domain are the base B and the number of qualitative values N for each sign:

Definition 7 ($P_{B,N}$ family of qualitative domains)

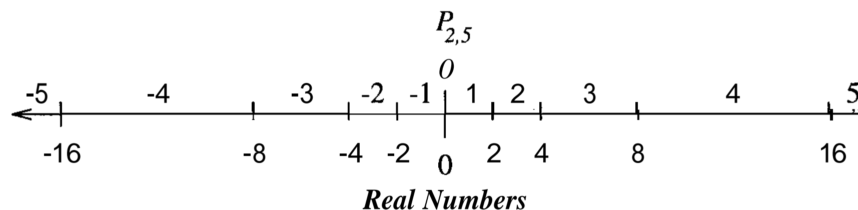
Given: $B \geq 2, N \geq 1$. B, N are integers.

$$P_{B,N}(x) = \begin{cases} 0 & \text{if } x = 0 \\ sign(x) \cdot 1 & \text{if } 0 < |x| \leq B \\ sign(x) \cdot \lceil \log_B |x| \rceil & \text{if } B < |x| \leq B^{N-1} \\ sign(x) \cdot N & \text{if } B^{N-1} < |x| \end{cases}$$

$P_{B,N}(x)$ is abbreviated as $\mathcal{P}(x)$ in formulas to save space.

The function is fairly straightforward. The reason that $0 < |x| \leq B$ is treated separately is to account for the sub-interval $0 < |x| < 1$, in which logarithms are negative.

As an example, if $B = 2$ and $N = 5$, the qualitative domain has the following definition:



The size of a qualitative domain $P_{B,N}$ is $2N + 1$.

Disregarding the qualitative values that go to $\pm\infty$, the "coverage" of $P_{2,5}$ is only 32 (from -16 to 16.) In contrast, $P_{10,7}$ covers a range of 2 million using only

four additional qualitative values. Thus, the parameters can be modified to suit the particular system they describe.

The inverse relation on $P_{B,N}$ is defined as follows.

Definition 8 (The inverse relation $P_{B,N}^{-1}$)

Given: $B \geq 2, N > 0, N = \lfloor N \rfloor$

$$P_{B,N}^{-1}(q) = \begin{cases} 0 & \text{if } q = 0 \\ sign(q) \cdot (0, B] & \text{if } |q| = 1 \\ sign(q) \cdot \left(B^{|q|-1}, B^{|q|}\right] & \text{if } 1 < |q| < N \\ sign(q) \cdot \left(B^{N-1}, +\infty\right) & \text{if } |q| = N \end{cases}$$

$P_{B,N}^{-1}(x)$ is abbreviated as $\mathcal{P}^{-1}(x)$ in formulas for brevity.

$P_{B,N}^{-1}(q)$ is defined as an interval over X for three of the four cases, making it a relation with respect to $\Re$.[1]

A proof of the inverse relation follows. It will use the following extended definition of $P_{B,N}$ and $P_{B,N}^{-1}$ over subsets of the real numbers and $\{-N, ..., N\}$:

Definition 9 ($P_{B,N}$ and $P_{B,N}^{-1}$ over sets)

Given $X \subseteq$ X and Q $\subseteq \{-N, ..., N\}$,

$$\mathcal{P}(X) = \{ q \mid x \in X \wedge \mathcal{P}(x) = q\}$$

$$\mathcal{P}^{-1}(Q) = \{ x \mid q \in Q \wedge x \in \mathcal{P}^{-1}(q)\}$$

Theorem 1 ($P_{B,N}^{-1}$ is the inverse relation of $P_{B,N}$)

Proving that $P_{B,N}^{-1}$ is the inverse relation for $P_{B,N}$ involves showing that a real number is associated with a qualitative value if and only if that qualitative value is associated

[1] $P_{B,N}^{-1}$ is, however, a function with respect to the domain of 4-tuples ({open, closed}, $\Re$, {open, closed}, $\Re$).

with the real number:

$$\forall q \in \{-N, ..., N\}, \forall a \in \Re \ : a \in \mathcal{P}^{-1}(q) \Leftrightarrow P(a) = q.$$

The reader will note that the definitions of $P_{B,N}$ and $P_{B,N}^{-1}$ both have four parts. For three parts the two definitions are identical: $x = 0$ and Q = 0, $0 < |x| \leq B$ and $|Q| = 1$, and $|x| > B^{N-1}$ and $|Q| = N$. The remaining case requires a little bit of work to show the inverse relationship:

1. Given an integer q, $1 < |q| < N$. Prove that $\mathrm{P}(\mathcal{P}^{-1}(q)) = q$.

   (a) Let $x = \mathcal{P}^{-1}(q) = sign(q) \cdot \left(B^{|q|-1}, B^{|q|}\right]$.

   (b) $|x| = \left(B^{|q|-1}, B^{|q|}\right]$ (Recall that $B > 0$.) Note that this implies that $sign(x) = sign(q)$.

   (c) Thus the range gives the inequality: $B^{|q|-1} < |x| \leq B^{|q|}$.

   (d) $1 < |q| \Rightarrow B < B^{|q|-1}$.

   (e) $|q| < N \Rightarrow B^{|q|} \leq B^{N-1}$.

   (f) Therefore, case three in the definition of $P_{B,N}$ applies:

$$\mathcal{P}(()x) = sign(x) \cdot \lceil \log_B |x| \rceil$$

   (g) Substituting the value of x into the equation:

$$\begin{aligned}
\mathcal{P}(()x) &= sign(x) \cdot \left\lceil \log_B \left| \left(B^{|q|-1}, B^{|q|}\right] \right| \right\rceil \\
&= sign(q) \cdot \lceil (|q| - 1, |q|] \rceil \\
&= sign(q) \cdot (|q| - 1, |q|] \\
&= sign(q) \cdot |q| \\
&= q
\end{aligned}$$

∎ (The third and fourth lines use the fact that q is an integer.)

2. Given a real number $x, B < |x| \leq B^{N-1}$. Prove that $x \in \mathcal{P}^{-1}(P(x))$.

   (a) Let $Q = P(x) = sign(x) \cdot \lceil \log_B |x| \rceil$.

   (b) $|q| = \lceil \log_B |x| \rceil$ (since $B > 0$). Also, $sign(q) = sign(x)$.

   (c) Substituting the given bounds on $|x|$:

$$\begin{aligned} B &< |x| \leq B^{N-1} \\ \log_B B &< \log_B |x| \leq \log_B B^{N-1} \\ 1 &< \log_B |x| \leq N-1 \\ \lceil 1 \rceil &< \lceil \log_B |B| \rceil \leq \lceil N-1 \rceil \\ 1 &< |q| \leq N-1 \end{aligned}$$

   (d) Therefore, case three in the definition of $P_{B,N}^{-1}$ applies:

$$\mathcal{P}^{-1}(q) = sign(q) \cdot \left(B^{|q|-1}, B^{|q|}\right]$$

   (e) In order to prove the postulate, we need to show that $x \in \mathcal{P}^{-1}(q)$. Since $sign(q) = sign(x)$, we can reduce the task to comparing $|x|$ to the range $\left(B^{|q|-1}, B^{|q|}\right]$:

$$\begin{aligned} B^{|q|-1} &< |x| \leq B^{|q|} \\ B^{\lceil \log_B |x| \rceil - 1} &< |x| \leq B^{\lceil \log_B |x| \rceil} \\ \lceil \log_B |x| \rceil - 1 &< \log_B |x| \leq \lceil \log_B |x| \rceil \end{aligned}$$

   (f) The left-hand inequality is

$$\begin{aligned} \lceil \log_B |x| \rceil - 1 &< \log_B |x| \\ \lceil \log_B |x| \rceil - \log_B |x| &< 1 \end{aligned}$$

   By definition, the ceiling of a number y is y if it is an integer, or the smallest integer greater than y if a non-integer. Thus the difference between y and its ceiling must be strictly less than 1, and the above inequality is valid. ∎

(g) The right-hand inequality is

$$\log_B |x| \leq \lceil \log_B |x| \rceil$$

Revisiting the definition of ceiling, we note that the ceiling of y is either y itself or some number slightly greater than y. This inequality is valid also, completing the proof. ∎

5.3 A Qualitative Calculus for $P_{B,N}$

This section defines the concept of a *qualitative function* in $P_{B,N}$ and applies this definition to some standard functions to build an algebra over $P_{B,N}$.

Definition 10 (Qualitative Function)

Given a function $f(x, y, ...)$ and qualitative values $q, r, ... \in P_{B,N}$, the *qualitative function* for f is annotated as $\lfloor f \rfloor$ and defined as

$$\lfloor f \rfloor(q, r, ...) = \{a | x \in \mathcal{P}^{-1}(q) \wedge y \in \mathcal{P}^{-1}(r) \wedge ... \wedge \mathcal{P}(f(x, y, ...)) = a\}$$

Theorem 2 (Negation)

The negation of a qualitative value is the negation of its numeric symbol:

$$\lfloor - \rfloor q = -q$$

Proof :			**Reason**
Given:		$-N \leq q \leq N$	
$q = 0 \Rightarrow \lfloor - \rfloor q$	$=$	$\mathcal{P}(-(0)) = 0 = -q$	Def. 7

$$
\begin{aligned}
|q| = 1 \Rightarrow \lfloor - \rfloor q &= \mathcal{P}\left(-(sign(q) \cdot (0, B])\right) \\
&= \mathcal{P}\left(sign(-q) \cdot (0, B]\right) \\
&= -q && \text{Def. 7} \\
1 < |q| < N \Rightarrow \lfloor - \rfloor q &= \mathcal{P}\left(-(sign(q) \cdot \left(B^{|q|-1}, B^{|q|}\right])\right) && \text{Def. 8} \\
&= \mathcal{P}\left(sign(-q) \cdot \left(B^{|q|-1}, B^{|q|}\right]\right) \\
&= -q && \text{Def. 7} \\
|q| = N \Rightarrow \lfloor - \rfloor q &= \mathcal{P}\left(-(sign(q) \cdot \left(B^{N-1}, +\infty\right))\right) && \text{Def. 8} \\
&= \mathcal{P}\left(sign(-q) \cdot \left(B^{N-1}, +\infty\right)\right) \\
&= -q && \text{Def. 7}
\end{aligned}
$$

∎

Before proceeding to the mathematical proofs, it would be helpful to construct some lemmas. A lemma that will be useful in the proofs in this section is the preservation of commutivity in $P_{B,N}$:

Theorem 3 (Preservation of Commutivity)

$$\lfloor f \rfloor(q, r) = \lfloor f \rfloor(r, q)$$

Proof : **Reason**

(5.1) *Given:* $f : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ is commutative.

$q, r \in \{-N, \ldots, N\}$

$$
\begin{aligned}
\lfloor f \rfloor(q, r) &= \{a \mid x \in \mathcal{P}^{-1}(q) \wedge y \in \mathcal{P}^{-1}(r) \wedge \mathcal{P}(f(x, y)) = a\} && \text{Def. 10} \\
&= \{a \mid x \in \mathcal{P}^{-1}(q) \wedge y \in \mathcal{P}^{-1}(r) \wedge \mathcal{P}(f(y, x)) = a\} && (5.1) \\
&= \lfloor f \rfloor(r, q) && \text{Def. 10}
\end{aligned}
$$

∎

The following lemma will be useful in deriving qualitative addition. It proves that the addition of two positive values in $P_{B,N}$ results in the larger value or a value one greater.

Theorem **4** (Sum **of** two positive values in $P_{B,N}$)

$$\forall q, r : 0 < q \leq r < N \Rightarrow \mathcal{P}\left(B^q + B^r\right) = r + 1$$

	Proof :			Reason
	$\mathcal{P}^{-1}(r+1)$	$=$	$(B', B^{r+1}]$	Def. 8
(5.2)		$\Rightarrow$	$B' < B^q + B^r$	$(B \geq 2)$
	B^{r+1}	$=$	$B \cdot B'$	
		$\geq$	$2B'$	$(B \geq 2)$
		$\geq$	$B' + B'$	
(5.3)		$\geq$	$B^q + B'$	$(q \leq r)$
	$\mathcal{P}\left(B^q + B^r\right)$	$=$	$r + 1$	(5.2), (5.3), Def. 7
			∎	

The following lemma is also used for the addition proof. It proves that if a positive qualitative number is subtracted from a larger qualitative number, then the result is equal to the larger number or one less.

Theorem **5** (Difference **of** two positive values in $P_{B,N}$)

$$\forall q, r \in \mathcal{I} : 0 < q < r < N \Rightarrow \mathcal{P}\left(B^r - B^q\right) = r$$

	Proof :			Reason
	$\mathcal{P}^{-1}(r)$	$=$	$\left(B^{r-1}, B^r\right]$	Def. 8
	B^r	$=$	$B \cdot B^{r-1}$	
		$\geq$	$2B^{r-1}$	$(B \geq 2)$
		$\geq$	$B^{r-1} + B^{r-1}$	
	$B^r - B^q$	$\geq$	$B^{r-1} + B^{r-1} - B^q$	
(5.4)		$>$	B^{r-1}	$(q \leq r-1)$
(5.5)	$B^r - B^q$	$<$	B^r	$(B \geq 2,\ q > 0)$
(5.6)	$\mathcal{P}\left(B' - B^q\right)$	$=$	r	(5.4), (5.5), Def. 7
			∎	

The following lemma shows that negation is distributive over multiplication in $P_{B,N}$.

Theorem 6 ($q\lfloor\cdot\rfloor r = -q\lfloor\cdot\rfloor -r$)

Inspection of definition 8 shows that each qualitative value q can be written as $sign(q) \cdot A_q$, where A, is an interval on $[0,+\infty)$. $sign(q)$ determines the sign of q and A, the range of magnitudes.

Proof :			**Reason**
$q\lfloor\cdot\rfloor r$	$=$	$\mathcal{P}(\mathcal{P}^{-1}(q) \cdot \mathcal{P}^{-1}(r))$	Def. 10
$q\lfloor\cdot\rfloor r$	$=$	$\mathcal{P}(sign(q) \cdot A_q \cdot sign(r) \cdot A_r)$	
	$=$	$\mathcal{P}(sign(q) \cdot sign(r) \cdot A_q \cdot A_r)$	
	$=$	$\mathcal{P}(-1 \cdot -1 \cdot sign(q) \cdot sign(r) \cdot A_q \cdot A_r)$	
	$=$	$\mathcal{P}(sign(-q) \cdot A_q \cdot sign(-r) \cdot A_r)$	
	$=$	$-q\lfloor\cdot\rfloor -r$	Def. 10
		∎	

Although it might at first seem that

$$-q\lfloor\cdot\rfloor r = q\lfloor\cdot\rfloor -r$$

can be proven by preservation of commutivity (Thm. **3**), such a proof would need the lemma $\lfloor-\rfloor q = -1\lfloor\cdot\rfloor q$, which is unfortunately not valid under $P_{B,N}$. This is a deficiency in $P_{B,N}$ that would be remedied by having a special value for multiplicative identity (*i.e.*, $\mathcal{P}^{-1}(1) = 1$) just as $P_{B,N}$ has for additive identity. This improvement is discussed in chapter 8.

Theorem 7 (Distributivity of Negation)

$$-q\lfloor\cdot\rfloor r = q\lfloor\cdot\rfloor -r = -(q\lfloor\cdot\rfloor r)$$

(Refer to Thm. 6 for description of A, and A_r.)

Proof :			**Reason**
$-q\lfloor\cdot\rfloor r$	$=$	$\mathcal{P}(\mathcal{P}^{-1}(-q) \cdot \mathcal{P}^{-1}(r))$	Def. 10

$$
\begin{aligned}
&= P\,(sign(-q)\cdot A_q \cdot sign(r)\cdot A_r) \\
&= P\,(-1 \,.\, sign(q)\cdot A_q \cdot sign(r)\,.\,A_r) \\
&= P\,(sign(q)\cdot A_q \cdot -1\,.\,sign(r)\cdot A_r) \\
&= P\,(sign(q)\cdot A_q \cdot sign(-r)\,.\,A_r) \\
&= q \boxdot -r \quad \blacksquare && \text{Def. 10} \\
&= P\,(-(sign(q)\,.\,A_q \cdot sign(r)\cdot A_r)) \\
&= \boxminus \mathcal{P}\,(sign(q)\,.\,A_q \,.\, sign(r)\cdot A_r) && \text{Def. 10} \\
&= -(q \boxdot -r) \quad \blacksquare && \text{Thm. 2, Def. 10}
\end{aligned}
$$

5.3.1 Addition

The rules for addition in $P_{B,N}$ are given by table 1.

Table 1 (Addition Summarized)

	q	r	$q \boxplus r$	Line Ref.
1	$-N$	$-N,\ldots,-1$	$-N$	(5.22), (5.28), (5.33)
2	$-N+1,\ldots,-1$	$q,\ldots,-1$	$q-1, q$	(5.15), (5.18), (5.24)
3	$-N,\ldots,-1$	0	q	(5.9), (5.11), (5.13)
4	$-N,\ldots,-3$	$1,\ldots,-q-2$	$q, q+1$	(5.20), (5.26)a, (5.29)a, (5.34)
5	$-N,\ldots,-2$	$-q-1$	$q,\ldots,-1$	(5.20)($q=2$), (5.26)b, (5.29)b
6	$-N,\ldots,-1$	$-q$	$q,\ldots,r$	(5.16), (5.25)c, (5.26)c, (5.31)
7	$-N+1),\ldots,-1$	$-q+1$	$1,\ldots,r$	(5.19)($q=2$), (5.25)b, (5.30)b
8	$-N+2,\ldots,-1$	$-q+2,\ldots,N$	$r-1, r$	(5.19)($q>2$), (5.25)a, (5.30)a, (5.35)
9	0	$0,\ldots,N$	r	(5.7), (5.8), (5.10), (5.12)
10	$1,\ldots,N-1$	$q,\ldots,N-1$	$r, r+1$	(5.14), (5.17), (5.23)
11	$1,\ldots,N$	N	N	(5.21), (5.27), (5.32)

Theorem 8 (Addition)

Table 1 conforms to the definition of qualitative functions.

Proof : **Reason**

Given: $-N < n \leq m < 0 < q \leq r < N$

$$
\begin{aligned}
0 \boxplus 0 &= P\left([0,0] + [0,0]\right) && \text{Def. 10, Def. 8} \\
&= \mathcal{P}\left([0,0]\right) \\
(5.7) \qquad &= 0 && \text{Def. 7}
\end{aligned}
$$

$$
\begin{aligned}
0 \boxplus 1 &= \\
1 \boxplus 0 &= \mathcal{P}\left((0,B] + [0,0]\right) && \text{Thm. 3, Def. 10, Def. 8} \\
&= \mathcal{P}\left((0,B]\right) \\
(5.8) \qquad &= 1 && \text{Def. 7}
\end{aligned}
$$

$$
\begin{aligned}
\text{OM-1} &= \\
-1 \boxplus 0 &= -(1 \boxplus 0) && \text{Thm. 3, Thm. 7} \\
(5.9) \qquad &= -1 && (5.8)
\end{aligned}
$$

$$
\begin{aligned}
0 \boxplus q &= \\
q \boxplus 0 &= \mathcal{P}\left((B^{q-1}, B^{q}] + [0,0]\right) && \text{Thm. 3, Def. 10, Def. 8} \\
&= P\left((B^{q-1}, B^{q}]\right) \\
(5.10) \qquad &= q && \text{Def. 7}
\end{aligned}
$$

$$
\begin{aligned}
0 \boxplus m &= \\
m \boxplus 0 &= -(-m \boxplus 0) && \text{Thm. 3, Thm. 7} \\
(5.11) \qquad &= m && (5.10)
\end{aligned}
$$

$$
\begin{aligned}
0 \boxplus N &= \\
N \boxplus 0 &= P\left(\left(B^{N-1}, +\infty\right) + [0,0]\right) && \text{Thm. 3, Def. 10, Def. 8} \\
&= \mathcal{P}\left(\left(B^{N-1}, +\infty\right)\right) \\
(5.12) \qquad &= N && \text{Def. 7}
\end{aligned}
$$

$$
\begin{aligned}
0 \boxplus -N &= \\
-N \boxplus 0 &= -(N \boxplus 0) && \text{Thm. 3, Thm. 7} \\
(5.13) \qquad &= -N && (5.12)
\end{aligned}
$$

$$
\begin{aligned}
1 \boxplus 1 &= \mathcal{P}\left((0,B] + (0,B]\right) && \text{Def. 10, Def. 8} \\
&= \mathcal{P}\left((0,2B]\right) \\
(5.14) \qquad &= \{1,2\} && \text{Def. 7}
\end{aligned}
$$

$$
-1 \boxplus -1 = -(1 \boxplus 1) \qquad \text{Thm. 7}
$$

$$\begin{array}{rcll} (5.15) & = & \{-2,-1> & (5.14) \end{array}$$

$$\begin{array}{rcll} 1\boxplus -1 & = & & \\ -1\boxplus 1 & = & \mathcal{P}([B,0) + (0,B]) & \text{Thm. } \mathbf{3}, \text{ Def. 10, Def. 8} \\ & = & \mathcal{P}((-B,B)) & \\ (5.16) & = & \{-1,0,1\} & \text{Def. 7} \end{array}$$

$$\begin{array}{rcll} 1\boxplus q & = & & \\ q\boxplus 1 & = & P((B^{q-1},B^{q}] + (0,B]) & \text{Thm. } \mathbf{3}, \text{ Def. 10, Def. 8} \\ & = & P((B^{q-1},B^{q}+B]) & \\ (5.17) & = & \{q,q+1\} & \text{Def. 7} \end{array}$$

$$\begin{array}{rcll} -1\boxplus m & = & & \\ m\boxplus -1 & = & -(-m\boxplus 1) & \text{Thm. } \mathbf{3}, \text{ Thm. 7} \\ (5.18) & = & \{-m,-m+1\} & (5.17) \end{array}$$

$$\begin{array}{rcll} -1\boxplus q & = & & \\ q\boxplus -1 & = & \mathbf{P}((B^{q-1},B^{q}] + [B,0)) & \text{Thm. } \mathbf{3}, \text{ Def. 10, Def. } \mathbf{8} \\ & = & P((B^{q-1}-1,B^{q})) & \\ (5.19) & = & \{q-1,q\} & \text{Def. 7,}(q > 1) \end{array}$$

$$\begin{array}{rcll} 1\boxplus m & = & & \\ m\boxplus 1 & = & -(-m\boxplus -1) & \text{Thm. } \mathbf{3}, \text{ Thm. 7} \\ & = & -\{-m-1,-m\} & (5.19) \\ (5.20) & = & \{m,m+1\} & \end{array}$$

$$\begin{array}{rcll} 1\boxplus N & = & & \\ N\boxplus 1 & = & P\left(\left(B^{N-1},+\infty\right) + (0,B]\right) & \text{Thm. } \mathbf{3}, \text{ Def. 10, Def. 8} \\ & = & \mathcal{P}\left(\left(B^{N-1},+\infty\right)\right) & \\ (5.21) & = & N & \text{Def. 7} \end{array}$$

$$\begin{array}{rcll} -1\boxplus -N & = & & \\ -N\boxplus -1 & = & -(N\boxplus 1) & \text{Thm. } \mathbf{3}, \text{ Thm. 7} \\ (5.22) & = & -N & (5.21) \end{array}$$

$$\begin{array}{rcll} q\boxplus r & = & & \end{array}$$

$$\begin{aligned} r \boxplus q &= \mathcal{P}\left((B^{r-1}, B'] + [-B^{-q}, -B^{-q-1})\right) && \text{Thm. 3, Def. 10, Def. 8} \\ &= P\left((B^{r-1} + B^{q-1}, B^{r} + B^{q}]\right) \\ \text{(5.23)} \quad &= \{r, r+1\} && \text{Thm. 4} \end{aligned}$$

$$\begin{aligned} m \boxplus n &= \\ n \boxplus m &= -(-m \boxplus -n) && \text{Thm. 3, Thm. 7} \\ \text{(5.24)} \quad &= \{m-1, m\} && \text{(5.23)} \end{aligned}$$

$$\begin{aligned} r \boxplus m &= \\ m \boxplus r &= P\left([-B^{-m}, -B^{-m-1}) + [-B^{-r}, -B^{-r-1})\right) && \text{Thm. 3, Def. 10, Def. 8} \\ &= \mathcal{P}\left((B^{r-1} - B^{-m}, B^{r} - B^{-m-1})\right) \\ \text{(5.25)} \quad &= \begin{cases} \{r-1, r\} & \text{if } -m < r-1 \\ \{1, \ldots, r\} & \text{if } -m = r-1 \\ \{m, \ldots, r\} & \text{if } -m = r \end{cases} && \text{Thm. 5} \end{aligned}$$

$$\begin{aligned} q \boxplus n &= \\ n \boxplus q &= -(-n \boxplus -q) && \text{Thm. 3, Thm. 7} \\ &= -\begin{cases} \{-n-1, -n\} & \text{if } q < -n-1 \\ \{1, \ldots, -n\} & \text{if } q = -n-1 \\ \{-q, \ldots, -n\} & \text{if } q = -n \end{cases} && \text{(5.25)} \\ \text{(5.26)} \quad &= \begin{cases} \{n, n+1\} & \text{if } q < -n-1 \\ \{n, \ldots, -1\} & \text{if } q = -n-1 \\ \{n, \ldots, q\} & \text{if } q = -n \end{cases} \end{aligned}$$

$$\begin{aligned} q \boxplus N &= \\ N \boxplus q &= P\left(\left(B^{N-1}, +\infty\right) + (B^{q-1}, B^{q}]\right) && \text{Thm. 3, Def. 10, Def. 8} \\ &= P\left(\left(B^{N-1} + B^{q-1}, +\infty\right)\right) \\ \text{(5.27)} \quad &= N && \text{Def. 7} \end{aligned}$$

$$\begin{aligned} m \boxplus -N &= \\ -N \boxplus m &= -(N \boxplus -m) && \text{Thm. 3, Thm. 7} \\ \text{(5.28)} \quad &= -N && \text{(5.27)} \end{aligned}$$

$$\begin{aligned} q \boxplus -N &= \\ -N \boxplus q &= P\left(\left(-\infty, B^{N-1-1}\right) + (B^{q-1}, B^{q}]\right) && \text{Thm. 3, Def. 10, Def. 8} \\ &= P\left(\left(-\infty, -B^{N-1} + B^{q}\right)\right) \end{aligned}$$

$$\begin{array}{llll}
(5.29) & = & \begin{cases} \{-N,-N+1\} & \text{if } q < N-1 \\ \{-\mathrm{N},\ldots,-1\} & \text{if } q = N-1 \end{cases} & \text{Thm. 5} \\
m \boxplus N & = & & \\
N \boxplus m & = & -(-N \boxplus -m) & \text{Thm. 3,Thm. 7} \\
(5.30) & = & \begin{cases} \{N-1,N\} & \text{if } m > -N+1 \\ \{1,\ldots,\mathrm{N}\} & \text{if } m = -N+1 \end{cases} & (5.29) \\
-N \boxplus N & = & & \\
N \boxplus -N & = & P\left(\left(B^{N-1},+\infty\right) + \left(-\infty,B^{N-1-1}\right)\right) & \text{Thm.,3,Def. 10,Def. 8} \\
 & = & \mathcal{P}((-\infty,+\infty)) & \\
(5.31) & = & \{-\mathrm{N},\ldots,\mathrm{N}\} & \text{Def. 7} \\
N \boxplus \mathrm{N} & = & P\left(\left(B^{N-1},+\infty\right) + \left(B^{N-1},+\infty\right)\right) & \text{Def. 10,Def. 8} \\
 & = & P\left(\left(2B^{N-1},+\infty\right)\right) & \\
(5.32) & = & N & \text{Def. 7} \\
-N \boxplus -N & = & -(N \boxplus N) & \\
(5.33) & = & -N & (5.32) \\
1 \boxplus -N & = & & \\
-N \boxplus 1 & = & P\left(\left(-\infty,B^{N-1-1}\right) + (0,B]\right) & \text{Thm. 3,Def. 10,Def. 8} \\
 & = & P\left(\left(-\infty,-B^{N-1} + B\right)\right) & \\
(5.34) & = & \{-N,-N+1\} & \text{Thm. 5} \\
-1 \boxplus N & = & & \\
N \boxplus -1 & = & -(-N \boxplus 1) & \text{Thm. 3,Thm. 7} \\
(5.35) & = & \{\mathrm{N}-1,N\} & (5.34)
\end{array}$$

An interesting result of the proof is that addition is independent of the base B. The following example shows an addition table for the sub-family $P_{B,5}$, of which the previous example $P_{2,5}$ is a member.

Example 1 (Addition of $P_{B,5}$)

	-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-5	-5	-5	-5	-5	-5	-5,-4	-5,-4	-5,-4	-5...-1	-5...5
-4	-5	-5,-4	-5,-4	-5,-4	-5,-4	-4	-4,-3	-4,-3	-4...-1	-4...4	1...5
-3	-5	-5,-4	-4,-3	-4,-3	-4,-3	-3	-3,-2	-3...-1	-3...3	1...4	4,5
-2	-5	-5,-4	-4,-3	-3,-2	-3,-2	-2	-2,-1	-2...2	1,2,3	3,4	4,5
-1	-5	-5,-4	-4,-3	-3,-2	-2,-1	-1	-1,0,1	1,2	2,3	3,4	4,5
0	-5	-4	-3	-2	-1	0	1	2	3	4	5
1	-5,-4	-4,-3	-3,-2	-2,-1	-1,0,1	1	1,2	2,3	3,4	4,5	5
2	-5,-4	-4,-3	-3...-1	-2...2	1,2	2	2,3	2,3	3,4	4,5	5
3	-5,-4	-4...-1	-3...3	1,2,3	2,3	3	3,4	3,4	3,4	4,5	5
4	-5...-1	-4...4	1...4	3,4	3,4	4	4,5	4,5	4,5	4,5	5
5	-5...5	1...5	4,5	4,5	4,5	5	5	5	5	5	5

To provide a bit more motivation as to the efficacy of qualitative mathematics, let us explore an efficient representation of $P_{B,5}$. Since $P_{B,5}$ has 11 values, a number capable of representing any subset of these values would require 11 bits. The following implementation rounds to one and one-half bytes:

Example 2 (Hexadecimal Representation of $P_{B,5}$)

$P_{B,5}$ is a set of 11 values. The power set $2^{P_{B,5}}$ can be contained in 11 bits, which round up to 3 nybbles (half-bytes). The most-significant nybble can be used to represent the special values -5, 0, and 5, the middle nybble the bounded negative values, and the least-significant nybble the bounded positive values:

Special		**Negative**		**Positive**	
0	100	-1	010	1	001
-5	200	-2	020	2	002
5	400	-3	040	3	004
		-4	080	4	008

For example, the set of values $\{-4,-3,0,5\}$ would equal `5C0`.

Using this encoding scheme, we can implement addition as an 11 x 11 array:

Example 3 (Hexadecimal representation of table 1)

(Refer to example 2 for explanation of hexadecimal values.)

+	200	080	040	020	010	100	001	002	004	008	400
200	200	200	200	200	200	200	280	280	280	2F0	7FF
080	200	280	280	280	280	080	0C0	0C0	0F0	1FF	40F
040	200	280	0C0	0C0	0C0	040	060	070	177	00F	408
020	200	280	0C0	060	060	020	030	133	007	00C	408
010	200	280	0C0	060	030	010	111	003	006	00C	408
100	200	080	040	020	010	100	001	002	004	008	400
001	280	0C0	060	030	111	001	003	006	00C	408	400
002	280	0C0	070	133	003	002	006	006	00C	408	400
004	280	0F0	177	007	006	004	00C	00C	00C	408	400
008	2F0	1FF	00F	00C	00C	008	408	408	408	408	400
400	7FF	40F	408	408	408	400	400	400	400	400	400

Thus, the addition of whole ranges of numbers is reduced to referencing an array element.

5.3.2 Subtraction

Subtraction is derived from negation and addition. Given $q, r \in P_{B,N}$,

$$q \boxminus r = q \boxplus -r$$

The following table summarizes how one number in $P_{B,N}$ is subtracted qualitatively from another.

Table 2 (Subtraction)

	q	r	$q\boxminus r$
1	$-N$	$1,\ldots,N$	$-N$
2	$-N+1,\ldots,-1$	$1,\ldots,-q$	$q-1,q$
3	$-N,\ldots,-1$	0	q
4	$-N,\ldots,-3$	$q+2,\ldots,1$	$q,q+1$
5	$-N,\ldots,-2$	$q+1$	$q,\ldots,-1$
6	$-N,\ldots,-1$	q	$q,\ldots,r$
7	$-N+1,\ldots,-1$	$q-1$	$1,\ldots,r$
8	$-N+2,\ldots,-1$	$-N,\ldots,q-2$	$r-1,r$
9	0	$-N,\ldots,0$	r
10	$1,\ldots,N-1$	$-N+1,\ldots,-q$	$r,r+1$
11	$1,\ldots,N$	$-N$	N

5.3.3 Multiplication

Qualitative multiplication is defined as follows. The subsequent proof shows that this definition satisfies the concept of a qualitative function (Def. 10).

Definition 11 (Multiplication for $P_{B,N}$)

$$q\boxdot r = \begin{cases} 0 & \text{if } qr=0 \\ sign(qr)\cdot\{1,\ldots,r+1\} & \text{if } |q|=1 \\ sign(qr)\cdot\{1,\ldots,q+1\} & \text{if } |r|=1 \\ sign(qr)\cdot\{\min(|q|+|r|-1,N),\min(|q|+|r|,N)\} & \text{otherwise} \end{cases}$$

Theorem 9 (Multiplication)

Let: $0<|s|\leq N,\ 1<q<N,\ 1<r<N.$

Proof : **Reason**

			Reason
$-N\boxdot -N$	$=$	$N\boxdot N$	Thm. 6
	$=$	$\mathcal{P}\left(\left(B^{N-1},+\infty\right)\cdot\left(B^{N-1},+\infty\right)\right)$	Def. 10,Def. 8

$$
\begin{array}{llcll}
 & & = & P\left(\left(B^{2N-2},+\infty\right)\right) & \\
(5.36) & & = & N & \text{Def. 7}
\end{array}
$$

$$
\begin{array}{llcll}
 & N\lfloor\cdot\rfloor -N & = & -N\lfloor\cdot\rfloor N & \text{Thm. 3} \\
 & & = & -(N\lfloor\cdot\rfloor N) & \text{Thm. 7} \\
 & & = & -N & (5.36)
\end{array}
$$

$$
\begin{array}{llcll}
 & 0\lfloor\cdot\rfloor 0 & = & \mathcal{P}(0\cdot 0)=0 & \text{Def. 10,Def. 8,Def. 7} \\
 & s\lfloor\cdot\rfloor 0 & = & 0\lfloor\cdot\rfloor s & \text{Thm. 3} \\
 & & = & P\left(0\,.\,sign(s)\cdot|s|\right) & \text{Def. 10,Def. 8} \\
 & & = & 0 & \text{Def. 7}
\end{array}
$$

$$
\begin{array}{llcll}
 & -1\lfloor\cdot\rfloor -1 & = & 1\lfloor\cdot\rfloor 1 & \text{Thm. 6} \\
 & & = & \mathcal{P}\left((0,B]\cdot(0,B]\right) & \text{Def. 10,Def. 8} \\
 & & = & \mathcal{P}\left((0,B^2]\right) & \\
(5.37) & & = & \{1,2\} & \text{Def. 7}
\end{array}
$$

$$
\begin{array}{llcll}
 & 1\lfloor\cdot\rfloor -1 & = & -1\lfloor\cdot\rfloor 1 & \text{Thm. 7} \\
 & & = & -(1\lfloor\cdot\rfloor 1) & \text{Thm. 6} \\
 & & = & \{-2,-1\} & \text{(5.37),Thm. 2}
\end{array}
$$

$$
\begin{array}{llcll}
 & 1\lfloor\cdot\rfloor q & = & q\lfloor\cdot\rfloor 1 = -1\lfloor\cdot\rfloor -q = -q\lfloor\cdot\rfloor -1 & \text{Thm. 3,Thm. 6} \\
 & & = & P\left((0,\boldsymbol{B}]\cdot(\boldsymbol{B}^{q-1},\boldsymbol{B}^{q}]\right) & \text{Def. 10,Def. 8} \\
 & & = & P\left((0,B^{q+1}]\right) & \\
(5.38) & & = & \{1,\ldots,q+1\} &
\end{array}
$$

$$
\begin{array}{llcll}
 & -1\lfloor\cdot\rfloor q & = & q\lfloor\cdot\rfloor -1 = -q\lfloor\cdot\rfloor -1 = -1\lfloor\cdot\rfloor -q & \text{Thm. 3,Thm. 6} \\
 & & = & -(1\lfloor\cdot\rfloor q) & \text{Thm. 7} \\
 & & = & \{-q-1,\ldots,-1\} & (5.38)
\end{array}
$$

$$
\begin{array}{llcll}
 & q\lfloor\cdot\rfloor N & = & N\lfloor\cdot\rfloor q = -N\lfloor\cdot\rfloor -q = -q\lfloor\cdot\rfloor -N & \text{Thm. 3,Thm. 6} \\
 & & = & \mathcal{P}\left(\left(B^{N-1},+\infty\right)\cdot(B^{q-1},B^{q}]\right) & \text{Def. 10,defipbn} \\
 & & = & \mathcal{P}\left(\left(B^{N+q-2},+\infty\right)\right) & \\
(5.39) & & = & N & \text{Def. 7}
\end{array}
$$

$$
\begin{array}{llcll}
 & q\lfloor\cdot\rfloor -N & = & -N\lfloor\cdot\rfloor q = N\lfloor\cdot\rfloor -q = -q\lfloor\cdot\rfloor N & \text{Thm. 3,Thm. 6}
\end{array}
$$

$$
\begin{aligned}
&= -(q\lfloor\cdot\rfloor N) && \text{Thm. 7}\\
&= -N && \text{(5.39)}
\end{aligned}
$$

$$
\begin{aligned}
q\lfloor\cdot\rfloor r &= -q\lfloor\cdot\rfloor -r && \text{Thm. 6}\\
&= \mathcal{P}((B^{q-1}, B^{q}] \cdot (B^{r-1}, B'])) && \text{Def. 10,Def. 8}\\
&= \mathcal{P}((B^{q+r-2}, B^{q+r}])\\
\text{(5.40)} \qquad &= \begin{cases} \{q+r-1, q+r\} & \text{if } q+r \leq \mathrm{N} \\ \mathrm{N} & \text{if } q+r > N \end{cases} && \text{Def. 7}\\
q\lfloor\cdot\rfloor -r &= -r\lfloor\cdot\rfloor q && \text{Thm. 3}\\
&= -(r\lfloor\cdot\rfloor q) && \text{Thm. 7}\\
&= \begin{cases} \{-q-r, -q-r+1\} & \text{if } -q-r \geq -N \\ -N & \text{if } -q-r < N \end{cases} && \text{(5.40)}
\end{aligned}
$$

The following table shows multiplication for $P_{N,5}$.

Example 4 (Multiplication table for $P_{N,5}$)

	-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	5	5	5	5	1...5	0	-5...-1	-5	-5	-5	-5
-4	5	5	5	5	1...5	0	-5...-1	-5	-5	-5	-5
-3	5	5	5	4,5	1...4	0	-4...-1	-5,-4	-5	-5	-5
-2	5	5	5	3,4	1,2,3	0	-3...-1	-4,-3	-5,-4	-5	-5
-1	1...5	1...5	1...4	1,2,3	1,2	0	-2,-1	-3...-1	-4...-1	-5...-1	-5...-1
0	0	0	0	0	0	0	0	0	0	0	0
1	-5...-1	-5...-1	-4...-1	-3...-1	-2,-1	0	1,2	1,2,3	1...4	1...5	1...5
2	-5	-5	-5,-4	-4,-3	-3...-1	0	1,2,3	3,4	4,5	5	5
3	-5	-5	-5	-5,-4	-4...-1	0	1...4	4,5	5	5	5
4	-5	-5	-5	-5	-5...-1	0	1...5	5	5	5	5
5	-5	-5	-5	-5	-5...-1	0	1...5	5	5	5	5

5.3.4 Division

Division will not be proven formally here. The interested reader can use the strategy demonstrated for addition and multiplication or derive this definition directly from the multiplication table.

Definition 12 (Division)

$$\mathcal{P}\left(q \boxed{\cdot} r\right) = \begin{cases} 0 & \text{if } q = 0 \text{ and } r \neq 0 \\ -N, ..., N & \text{if } q = r = 0 \\ sign(qr) \cdot \{\max(1, |q| - |r|), ..., N\} & \text{if } |q| = N \text{ or } |r| = 1 \\ sign(qr) \cdot \left\{ \begin{array}{l} \max(1, |q| - |r|), \\ \max(1, |q| - |r| + 1|) \end{array} \right\} & \text{otherwise} \end{cases}$$

5.3.5 Relational Operators

Since qualitative values partition $\Re$ disjointly, two elements of qualitative values cannot be equal if the qualitative values are different. However, except for the point-interval zero, nothing can be said about $q \boxed{=} q$, since there exist values in $\mathcal{P}^{-1}(q)$ that are not equal to each other.

Table 3 (Equality)

$$q \boxed{=} r = \begin{cases} \text{false} & \text{i f q f r} \\ \text{true} & \text{if } q = r = 0 \\ \{\text{true, false}\} & \text{otherwise} \end{cases}$$

Inequality is simply the negation of equality:

Table 4 (Inequality)

$$q \boxed{\neq} r = \begin{cases} \text{true} & \text{if } q \neq r \\ \text{false} & \text{if } q = r = 0 \\ \{\text{true, false}\} & \text{otherwise} \end{cases}$$

The less-than operation $\boxed{<}$ is true if $q < r$ and false if $q > r$. In general, if $q = r$ then the value is unknown, with the exception of the point-interval zero:

Table 5 (Strictly Less Than)

$$q \boxed{<} r = \begin{cases} \text{true} & \text{if } q < r \\ \text{false} & \text{if } q > r \text{ or } q = r = 0. \\ \{\text{true, false}\} & \text{otherwise} \end{cases}$$

The remaining inequalities can be derived easily from $\boxed{<}$ by performing the following transformations:

$$\begin{aligned} q \boxed{<} r & \Leftrightarrow \neg(q \boxed{\geq} r) \\ & \Leftrightarrow r \boxed{>} q \\ & \Leftrightarrow \neg(r \boxed{\leq} q) \end{aligned}$$

5.4 Inverse Relations

The algorithm in the next chapter performs backward propagation on assumptions, so that the state can be modeled more completely. Backward propagation is accomplished by applying an *inverse relation* on each input.

The inverse relations are derived from the normal functions by solving for each input. For example, the inverse relations for the product $q \boxed{\cdot} r = s$ are

$$q = s \boxed{\div} r$$

$$r = s \boxed{\div} q$$

The results are intersected with the current values of q and r. Take, for example, the following set of values:

$$q = \{-3,-2,-1,0,1,2\}$$
$$r = \{0,1,2\}$$
$$s = \{-3,-2)$$

Calculating the inverse relations:

$$\begin{aligned} q_{new} &= q \bigcap s \boxed{\div} r \\ &= \{-3,-2,-1,0,1,2\} \bigcap \{-5,-4,-3,-2,-1\} \\ &= \{-3,-2,-1\} \\ r_{new} &= r \bigcap s \boxed{\div} q \\ &= \{0,1,2\} \bigcap \{-5,-4,-3,-2,-1,1,2,3,4,5\} \\ &= \{1,2\} \end{aligned}$$

These results correspond to the intuitive observations that only a negative can be combined with a positive to produce a negative (c.f. q), and zero cannot be a multiplicand of a non-zero product (c.f. r).

5.5 A Qualitative Calculus of Deviations

A contribution of this dissertation is the use of qualitative mathematics to describe deviations. The concept is similar to the guide words of HAZOP. Although digraph methods use a qualitative calculus to analyze flow deviations, it is important to note that the deviations are indistinguishable from normal relationships. That

is, influences are only considered deviations if they are qualified with a fault. For example, an influence of -1 if the valve is accidentally reversed is a deviation from the norm, but -1 is not the value of the deviation. It is simply the influence of one variable on another variable, the same as described for normal relationships. In contrast, the calculus developed in this chapter is specifically a calculus of deviations. The values characterize the value of a parameter in relation to its normal value.

Each qualitative value is a 2-tuple (V, D), where V is the qualitative domain representing the possible values a process variable can take and D is the qualitative domain representing the ways in which the variable can deviate from its correct value. D includes the special value 0, which indicates that the variable's value is correct. In order to avoid confusion, V will be referred to as the "value part" and D will be referred to as the "deviation part."

The value part of the quantitative domains for deviation analysis are the causality diagram function types (*i.e.*, the function ranges.) Recall from section 4.4.1 that three types are needed to translate an RSML specification: boolean, enumerated, and real. Another language may necessitate different quantitative domains.

In general, the values in a qualitative domain are chosen to distinguish as minimally as possible between meaningful alternatives [12]. This goal applies to both the value and deviation parts. The simplest calculus, sign algebra, was described above. Sign algebra as applied to the deviation part describes whether the value part is too high, too low, or correct.

The value-part functions are independent of the deviation part. That is, it does not matter whether a variable is a deviation when calculating what its value is. However, the deviation-part functions can be dependent on the value. Figure 5.2

$x + y$

	L	N	H
L	L	L	?
N	L	N	H
H	?	H	H

$x \times y$

		L			N			H		
		−	0	+	−	0	+	−	0	+
L	−	?	?	?	H	N	L	?	?	?
	0	?	L	L	H	N	L	H	H	?
	+	?	L	L	H	N	L	H	H	?
N	−	H	H	H	N	N	N	L	L	L
	0	N	N	N	N	N	N	N	N	N
	+	L	L	L	N	N	N	H	H	H
H	−	?	H	H	L	N	H	L	L	?
	0	?	H	H	L	N	H	L	L	?
	+	?	?	?	L	N	H	?	?	?

Table 5.2: Sign algebra of deviations. 'L,' 'N,' and 'H' represent values that are *too low, normal,* and *too high,* respectively.

shows the rules for addition and multiplication. Deviations due to addition are independent of the value part. However, the value part must be taken into account when calulating a product's deviation. Proofs of these calculations will be presented later in this chapter.

5.6 Deviation Formulae

This section defines the meaning of a deviation and derives formulae to calculate deviations for the functions introduced in section 4.4.1.

5.6.1 Numeric Functions

In order to formulate a calculus of deviations, it would be helpful to formally define what is meant by a deviation. For purposes of this calculus, a deviation of a numeric variable is defined to be the amount added to or subtracted from the correct value in order to obtain the actual value. For example, if the pressure should be

10 p.s.i. but is in actuality 7 p.s.i. then the deviation in pressure is **-3** p.s.i. The corresponding formula is

$$X_a = X_c + X_d, \tag{5.41}$$

where X is the variable, the subscript ***a*** means the actual value of X, ***c*** means the correct value of X, and d is the deviation in X's value.

Note that the deviation could be calculated in other ways. For example, X_d could be the ratio $\frac{X_a}{X_c}$. Under this paradigm, a value of $X_d = -0.5$ would mean that X_a is the opposite sign of and one-half the magnitude of X_c. While this formula is quite useful, X_d does not have a value when $X_c = 0$ and it is virtually meaningless when $X_a = 0$.

Equation (5.41) associates three values to each system variable. Since one value can be derived from the other two, the variable may be represented unambiguously by one of three pairs: $\langle X_a, X_c\rangle$, $\langle X_c, X_d\rangle$, or $\langle X_a, X_d\rangle$. Although any of these three alternatives would be appropriate, the calculus of deviations developed here is based on $\langle X_a, X_d\rangle$. The rationale is that X_d is most salient (this being a deviation analysis), and X_a is more salient than X_c since the analyst is presumably more interested in understanding deviations in the context of what actually happens than what should have happened.

Functions also conform to the above definition, *so* that replacing X with a function $f(X, Y, \ldots)$ yields the equation

$$f_a = f_c + f_d \tag{5.42}$$

The meaning of f_a and f_c are straightforward. They are defined in terms of the actual and correct values of independent variables X, Y, *etc.*, respectively:

$$f_a = f(X_a, Y_a, ..) \tag{5.43}$$

$$f_c = f(X_c, Y_c, ...) \tag{5.44}$$

However, since only the actual value and deviation are available for each variable, equation (5.41) is used to replace each parameter in (5.44) with the available input information:

$$\begin{aligned} f_c &= f(X_c, Y_c, ...) \\ &= f(X_a - X_d, Y_a - Y_d, ...) \end{aligned} \tag{5.45}$$

By rearranging equation (5.42) and substituting (5.43) and (5.45) for f_a and f_c, respectively, a general definition for function deviations can be derived solely in terms of the actual and deviation values of f's inputs:

$$\begin{aligned} f_d &= f_a - f_c \\ &= f(X_a, Y_a, ...) - f(X_a - X_d, Y_a - Y_d, ...) \end{aligned} \tag{5.46}$$

Equation (5.46) can be used as a template to derive the deviation functions for numeric functions. **A** simple example to begin with is addition. The following proof uses equations (5.46)) (5.43)) and (5.45) to show that the deviation of $X + Y$ is simply the sum of their deviations:

$$\begin{aligned} (X+Y)_d &= (X+Y)_a - (X+Y)_c \\ &= (X_a + Y_a) - ((X_a - X_d) + (Y_a - Y_d)) \\ &= X_d + Y_d \end{aligned} \tag{5.47}$$

Unary negation is as follows:

$$\begin{aligned} (-X)_d &= (-X)_a - (-X)_c \\ &= (-X_a) - (-(X_a - X_d)) \\ &= -X_a + X_a - X_d \\ &= -X_d \end{aligned} \tag{5.48}$$

Subtraction may be derived by using the results of addition and negation:

$$\begin{aligned}(X-Y)_d &= (X+(-Y))_d \\ &= X_d+(-Y)_d \\ &= X_d - Y_d \qquad (5.49)\end{aligned}$$

Thus far the proofs have shown that $f_d = f(X_d, Y_d, ...)$ for some functions. The following proof for multiplication shows that deviations can interact with each other as well as the actual values:

$$\begin{aligned}(XY)_d &= (XY)_a - (XY)_c \\ &= X_aY_a - (X_a - X_d)(Y_a - Y_d) \qquad (5.50) \\ &= X_aY_a - (X_aY_a - X_dY_a - X_aY_d + X_dY_d) \\ &= X_dY_a + X_aY_d - X_dY_d \qquad (5.51)\end{aligned}$$

Division is as follows:

$$\begin{aligned}\left(\frac{X}{Y}\right)_d &= \left(\frac{X}{Y}\right)_a - \left(\frac{X}{Y}\right)_c \\ &= \frac{X_a}{Y_a} - \frac{X_c}{Y_c} \\ &= \frac{X_a}{Y_a} - \frac{X_a - X_d}{Y_a - Y_d} \\ &= \frac{X_aY_a - X_aY_d - (X_aY_a - X_dY_a)}{Y_a(Y_a - Y_d)} \\ &= \frac{X_dY_a - X_aY_d}{Y_a(Y_a - Y_d)} \qquad (5.52)\end{aligned}$$

Reciprocal can be derived from quotient by setting $X_a = 1$ and $X_d = 0$:

$$\begin{aligned}\left(\frac{1}{Y}\right)_d &= \frac{(0)Y_a - (1)Y_d}{Y_a(Y_a - Y_d)} \\ &= \frac{Y_d}{Y_a(Y_a - Y_d)} \qquad (5.53)\end{aligned}$$

5.6.2 Boolean Algebra

Since there are only two values in the boolean domain, the concept of a deviation is limited to whether the value is correct or incorrect. Taking a logical "0" as being the special value *correct*, and hence logical "1" as a deviation, the following truth table shows the relationship between correct and actual values and deviations:

Correct (B_c)	**Actual** (B_a)	**Deviation** (B_d)
0	0	0
0	1	1
1	0	1
1	1	0

One can readily see that this table describes the exclusive-or operation. Thus, the boolean equivalent to equation (5.41) is

$$B_a = B_c \oplus B_d. \tag{5.54}$$

Given a boolean operator $p(B, C, ...)$, the following equations are the boolean equivalents of the numeric equations (5.42)–(5.46):

$$p_a = p_c \oplus p_d \tag{5.55}$$

$$= p(B_a, C_a, ...) \tag{5.56}$$

$$p_c = p(B_c, C_c, ...) \tag{5.57}$$

$$= p(B_a \oplus B_d, C_a \oplus C_d, ...) \tag{5.58}$$

$$p_d = p_a \oplus p_c \tag{5.59}$$

$$= p(B_a, C_a, ...) \oplus p(B_a \oplus B_d, C_a \oplus C_d, ...) \tag{5.60}$$

Negation is as follows:

$$(\neg B)_d = \neg B_a \oplus \neg B_a \oplus B_d$$

$$= \neg B_a \oplus \neg B_a \oplus B_d$$

$$= B_d \tag{5.61}$$

Intuitively, this equation states that if a boolean variable's value is incorrect (or correct), then its negation is also.

Logical conjunction is given by

$$(BC)_d = (B_aC_a) \oplus ((B_a \oplus B_d)(C_a \oplus C_d))$$

This form poses a problem for the forward algorithm presented in the next chapter. If B_a and B_d are both true, then $B, \oplus B_d$ is false. False means "no deviation" and the algorithm will not continue its search of $(BC)_d$. Factoring out $\oplus$ and rearranging the terms gives:

$$\begin{aligned} (BC)_d &= B_d(B_aC_a \vee \neg B_a(C_a \oplus C_d) \vee \\ & \quad C_d(B_aC_a \vee \neg C_a(B_a \oplus B_d) \end{aligned}$$

If B_a and B_d are both true, then the forward search algorithm will make the assumption that

$$(B_aC_a \vee \neg B_a(C_a \oplus C_d) = \text{True}$$

which reduces to $C_a = \text{True}$.

Substituting logical disjunction into **5.55** gives

$$(B \vee C)_d = (B_a \vee C_a) \oplus ((B_a \oplus B_d) \vee (C_a \oplus C_d))$$

Disjunction can likewise be rearranged to facilitate forward analysis:

$$\begin{aligned} (B \vee C)_d &= B_d(\neg B_a \neg C_a \vee B_a \neg(C_a \oplus C_d) \vee \\ & \quad C_d(\neg B_a \neg C_a \vee C_a \neg(B_a \oplus B_d) \end{aligned}$$

5.6.3 Relational Operators

Equations (5.46) and (5.55) both are used to determine the deviation equations for the numeric relational operators. Like disjunction and conjunction, the relational operators do not simplify beyond the given formulae:

$$
\begin{aligned}
(X = Y)_d &= (X_a = Y_a) \oplus (X_c = Y_c) \\
&= (X_a = Y_a) \oplus (X_a - X_d = Y_a - Y_d) \\
&= (X_a = Y_a)\neg(X_a - X_d = Y_a - Y_d) \vee \neg(X_a = Y_a)(X_a - X_d = Y_a - Y_d) \\
(X \neq Y)_d &= (\neg(X = Y))_d = (X = Y)_d \\
&= (X_a = Y_a)\neg(X_a - X_d = Y_a - Y_d) \vee \neg(X_a = Y_a)(X_a - X_d = Y_a - Y_d) \\
(X < Y)_d &= (X_a < Y_a) \oplus (X_c < Y_c) \\
&= (X_a < Y_a) \oplus (X_a - X_d < Y_a - Y_d) \\
(X > Y)_d &= (X_a > Y_a) \oplus (X_c > Y_c) \\
&= (X_a > Y_a) \oplus (X_a - X_d > Y_a - Y_d) \\
(X \leq Y)_d &= (\neg(X > Y))_d = (X > Y)_d \\
&= (X_a > Y_a) \oplus (X_a - X_d > Y_a - Y_d) \\
(X \geq Y)_d &= (\neg(X < Y))_d = (X < Y)_d \\
&= (X_a < Y_a) \oplus (X_a - X_d < Y_a - Y_d)
\end{aligned}
$$

5.6.4 Enumerated Types

An enumerated type is defined as a set of tokens over which the '<' operator has been defined (thus enumerating the tokens.) The following function can be used

to map from an enumerated type $\mathcal{E}$ to the integers:

$$M_{\mathcal{E}}(e) \quad = \quad \| \{ f \mid f \in \mathcal{E}, \{ <]\} \|$$

This function allows the deviation equations of the relational operators to be extended to enumerated types by first mapping the inputs to the integers. For example, given $e, f \in \mathcal{E}$,

$$(e = f)_d = (M_{\mathcal{E}}(e) = M_{\mathcal{E}}(f))_d$$

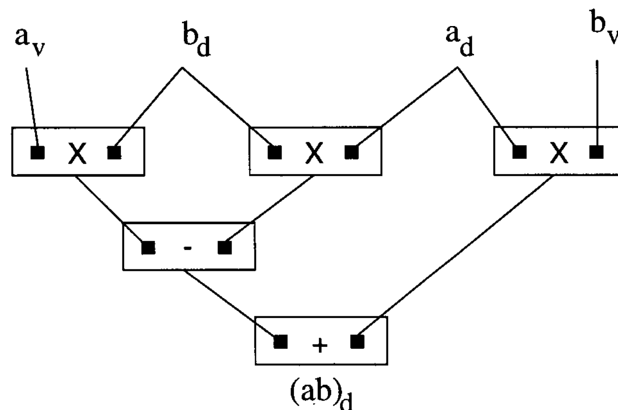
5.7 Application of Deviation Formulae to $P_{B,N}$

Now that the deviation formulae and the qualitative domain $P_{B,N}$ have been defined, they can be combined to create a qualitative calculus of deviations.

The deviation calculus could be extended by performing the kind of proof demonstrated in section *5.3* on the deviation formulae. However, a simpler solution is to extend the causality diagram to include the deviation part. For example, recalling that the deviation formula for multiplication is

$$(xy)_d = x_a y_a - (x_a - x_d)(y_a - y_d),$$

the causality diagram fragment would be constructed as follows:



This strategy has two advantages. First, it is a straightforward construction with no necessary extensions to the $P_{B,N}$ algebra. Second, the inverse relationship between deviations and the input values on which they depend can be exploited. For example,

$$
\begin{aligned}
x_a &= \frac{(xy)_d - y_a x_d + x_d y_d}{y_d} \\
y_a &= \frac{(xy)_d - x_a y_d + x_d y_d}{x_d}
\end{aligned}
$$

The inverse deviation relations are calculated in the same manner as for the value part. For the multiplication example,

$$
\begin{aligned}
x_d &= \frac{(xy)_d - x_a y_d}{y_a - y_d} \\
y_d &= \frac{(xy)_d - y_a x_d}{x_a - x_d}
\end{aligned}
$$

5.8 Assumptive Functions

The deviation analysis algorithm must often make assumptions in order to propagate deviations forward. The mathematical equivalent to this activity is termed the *assumptive function.* The goal is to propagate a deviation, so the function assumes that the deviation does propagate (a non-zero deviation part) and calculates the inverse relation accordingly.

5.8.1 Addition

The deviation function for addition is

$$(x + y)_d = x_d + y_d.$$

In order to propagate $x_d \# 0$ to produce $(z + y)_d \neq 0$ the sum must be non-zero. Thus, remove $-x_d$ from the set of possibilities for y_d. For example, if $x_d = \{1,2,3\}$, and $y_d = \{-1,0,1\}$, then the new value for y_d is $\{0,1\}$, making $(x \boxplus y)_d = \{1,2,3,4\}$.

5.8.2 Subtraction

Subtraction is similar to addition, except values from x_d are removed from y_d's set of possibilities. Given the same example as for addition, the new $y_d = \{-1,0\}$.

5.8.3 Multiplication

Recall that

$$(xy)_d = x_a y_a - (x_a - x_d)(y_a - y_d)$$

Now assume that x_d is non-zero and we want to make $(xy)_d$ non-zero. The terms of which x_d is a part must be non-zero too. First of all, $(x, - x_d) \neq 0$. **As** discussed above, the set of possible values for x_d must be removed from that for x_a. For example, if $x_d > 0$, then $x_a \leq 0$. Proceeding to the product, $(y_a - y_d)$ cannot be zero, since this would cancel the effect of x_d. Finally, $x_a y_a$ cannot be the same value as its subtrahend. In the most general case, this entails setting $x_a y_a$ to zero in order to propagate the deviation. However, additional constraints on the actual and deviation values of x and y can loosen the constraint on $x_a y_a$.

Since multiplication is commutative, the same rules apply to propagating y_d.

5.9 States

The algorithm presented in the next chapter performs a forward search based on qualitative values of nodes in a causality diagram. The node–value pairs are collected into sets referred to as *states* in this thesis. Each state describes the values of some subset of nodes in the diagram at a particular instant of time. Since the nodes are associated with either the system variables or some part of a system variable's definition, then the state also describes some portion of the system state. Generally speaking, each state is consistent with some subset of the system states Q. Note that the system variables have quantitative domains whereas the bindings in a state have corresponding qualitative domains. Since many qualitative values can map to a single qualitative value, a single state binding can represent many state bindings.

Deviation analysis begins with an initial state, composed of the analyst's starting assumptions. Assumptions are in the form of node value assignments, or *bindings.* Using these initial bindings, the algorithm traces structural relations in the causality graph, adding to the state's set of bindings.

Deviation analysis also involves tracing the causality graph's sequential relations. Normally the sequential relationship is traced forward (from cause-to-effect) since deviation analysis is principally a forward analysis method. However, if a state contains assumptions, as the initial state does, the causal relationships are also traced backward one step to infer further bindings in the current state. This exercise is part of the algorithm discussed in chapter 6. What is relevent at this point is knowing that each state has a *next* state, which is in most cases calculated by following sequential relationships forward. A state logically implies its next state, *i.e.*, given a system

state S that is described by the bindings in qualitative state Q, it must be true that $Next(Q)$ describes all possible system states that immediately follow S.[2]

The activity of using known bindings and causality graph to derive new bindings is referred to as *propagation* in this thesis. A synonymous phrase that will be used in chapter 6 is that the node's value has been "updated" for a state. Please note that the update occurs in the state's data structure, *i.e.*, progress is made in analysis of a particular system state. The phrase does not mean that some system variable has been updated, *i.e.*, that the system has changed state. Changes are recorded as new states in deviation analysis.

As shown in chapter 3 and as further described in this chapter, sometimes additional assumptions need to be made in order to propagate deviations. When this is the case the assumptions are added to the existing state in the form of node bindings. This creates a new state whose bindings are a superset of the original state's bindings. The number of degrees of freedom in the variables is correspondingly fewer. The new state is referred to as a *derived state.* The state to which it added its assumptions is called the *base state.* A state that has no base state is called a *root state.* The bindings in a state that are not in its base state are called its *leaf bindings.*

Since more than one set of assumptions may be made about a state to propagate deviations, that state may have more than one derived state. Conversely, each derived state has only one base state. Therefore, the states form a heirarchy.

A qualitative state S can be viewed as a predicate on its variable bindings:

$$S = b_1 \text{ A } b_2 \text{ A } \ldots \text{ A } b_n,$$

[2]Note that the concept of "immediately following" is implicitly defined by the model. It may mean a microsecond or a minute.

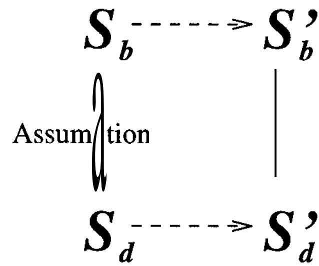


Figure 5.2: Example of how assumptions and sequential propagation relate.

where b_i is an assertion on a variable binding (*e.g.*, v = High.) S being true means that it is consistent with the state of the system at the time considered. The causality diagram (which the reader may recall is essentially a set of axioms) can be used to calculate S', which is the predicate describing what is true immediately subsequent to S being true. This task will be described in detail in section **6.3.1,** but briefly it is accomplished by first following sequential edges to build a minimal set of bindings for the next state and then following structural edges to obtain a full description of the state.

A derived state S_d is composed of the bindings of its base state S_b and its leaf bindings:

$$S_d = S_b \wedge b_1 \wedge \ldots \wedge b_n.$$

Thus it is trivially true that a base state implies its derived states. This observation may be combined with the fact that a state implies its next state, yielding a sort of lattice of implications, shown in figure 5.2. In this figure, forward sequential analysis produces state S'_b from S_b. Further analysis makes an assumption on the state S_b producing the derived state S_d. S'_d can be produced directly by a forward sequential analysis of S_d (including the bindings S_d inherits from S_b.) However, such an exercise involves some redundant calculations, since the leaf bindings in S_d often do not change the propagation of values in S_b. That is, some bindings in S'_d are the same as those in

```
function Contains-Deviation( S : Mode ) : Boolean
  begin
    if Deviations-In-Leaf(S) then
      return True;
    else if not Root(S) then
      return Contains_Deviation(Base(S));
    else
      return False;
  end;

function Deviation-In-Leaf( S : Mode) : Boolean
  begin
    for each binding B in S do
      if Is_Deviation(Value(B)) then
        return True;
    return False;
  end;
```

Figure 5.3: Algorithms for determining whether a mode contains any deviations.

S'_b. In fact, S'_d can be treated as a derived state of S'_b. Only the forward propagations that involve leaf bindings of S_d need to be included in S'_d. All other bindings can be found in S'_b.

The automated algorithm in the next chapter discontinues its search along a particular path if a state does not contain any deviations, since deviations cannot be propagated from normal values. For purposes of that algorithm, a state S is considered to "contain deviations" if and only if it contains deviations in its leaf bindings or its base state contains deviations. Figure 5.3 shows an algorithmic definition.

Chapter 6

A Forward Search Algorithm

As stated in chapter 2, the goal of this thesis is to develop a forward-search algorithm for the purpose of hazard analysis of computer-controlled systems. The goals of hazard analysis are to identify hazards, identify their causal factors, and evaluate risk [16]. Since a forward search works from cause to effect, a forward-searching hazard analysis would tackle the task of identifying potential hazards.

The algorithms in this chapter assume the following items are available:

- causality diagram,
- forward, backward, and conditional function mappings
- initial state,
- stopping criteria.

The causality diagram was discussed in chapter 4. The function mappings were discussed in chapter 5. The initial state is the starting place of the analysis.

Deviation analysis may be thought of as a type of symbolic execution, since the specification is used to propagate symbols representing classes of values (as defined by the calculus of deviations.) It may also be thought of as a limited theorem prover that does not prove an *a priori* postulate, but rather can provide a *posteriori* "proof" of circumstances leading from an initial state (the proof's antecedent) to a hazardous

```
procedure Semi-Automated
  var
    S : Mode;
  begin
    Push_Mode(Get_Initial_Mode_From_Analyst);
    while Stack not empty do
      begin
        S := Top(Stack);
        Display_Mode(S);
        case (Get-Command-From-Analyst) of
          Generate-Derived-Modes:
                Propagate_Possible_Deviations(S);
                Display_Derived_Modes(S);
          Analyze-Derived-Mode:
                Push-Mode(Get-Derived-Mode-From-Analyst(S));
          Analyze-Next-Mode:
                Push-Mode(Next(S));
          Back-Track:
                Pop(Stack);
        end;
  end;

procedure Push_Mode(S : Mode)
  begin
    Push(Stack, S);
    Propagate-Definite_Deviations(S);
  end;
```

Figure 6.1: A semi-automated search procedure.

state. As will be discussed, this proof may involve making assumptions in addition to the antecedent. The steps of the proof are composed using rules of the calculus (the qualitative functions) and the axioms (the relations in the causality diagram.)

The search may be directed by the analyst, in which case the deviation analysis is *semi-automated,* or it may be directed by a top-level search algorithm, in which case it is *fully automated.* The advantage of a semi-automated search is that the analyst, who invariably has more knowledge of the system than is represented by the specification, can often direct the search toward the most serious hazard scenarios.

The advantage of a fully-automated search is that a large number of scenarios can be generated and prioritized very quickly. The analyst can then scan a large number of scenarios, focusing on the ones that appear to be the most interesting. Both methods are profiled in this chapter.

6.1 Semi-Automated Analysis

The SemiAutomated search is guided by a stack of states. The stack represents states that have already been visited by the analyst. This means that deviations have already been propagated for these states (refer to the presentation of Propagate--DefiniteDeviations.) The procedure Push_State performs the push and analysis.

The SemiAutomated procedure begins by obtaining the initial state from the analyst. The state is passed to the procedure Push_State, where it is analyzed for definite deviations and pushed onto the stack. This is the extent of the initialization of the procedure and it then proceeds to the command loop.

When presented with a new state the analyst has two options: analyze the next state in sequence (*i.e.*, go forward a step) or generate derived states. Derived states are discussed in detail later in this chapter, but briefly they are based on the current state with additional assumptions in order to propagate deviations. Once the derived states have been generated, the analyst may choose one to analyze. **A** search may be concluded or postponed at any point by back-tracking, *i.e.*, popping states off the stack.

The command loop (while Stack not empty do) first pops the top state off the stack. It displays the state, including any auxiliary information obtained from

```
procedure Automated
  var
    Depth : integer;
    S : Mode;
  begin
    Append(Queue, Get-Initial_Mode_From_Analyst);
    while Queue not empty do
      begin
        S := Pop-Front(Queue);
        Propagate-Definite_Deviations(S);
        if ( Contains_Deviations(S)
             and Consistent(S)
             and Unique(S)
             and not Hazard(S) ) then
        begin
          if (Step(S) < MAX-STEPS) then
            Prepend(Queue, Next(S));
          if (Depth(S) < MAX-DEPTH) then
            begin
              Propagate-Possible-Deviations(S);
              for each child mode C of S do
                Append(Queue, C);
            end;
        end
      else if Hazard(S) then
        Append_Search_Path_To_Hazard_List(S);
  end;
```

Figure 6.2: A fully-automated search procedure.

PropagateDefiniteDeviations, such as whether it is inconsistent, whether it dead-ends, or if it contains a hazardous deviation (as defined by the stop criteria.) The analyst is then prompted for a command. If the command is to generate derived states, then it does so, displaying the results. If the command is to visit a derived state, then the user is prompted for the state to visit. The derived state is analyzed and pushed onto the stack. If the analyst gives the command to visit the next state in time, then that state is created if necessary (by Next), analyzed, and pushed onto the stack. The back-tracking command pops the top state off the stack so that the next

iteration of the command loop refers to the state analyzed just before the current one. The analysis concludes when all of the states are popped off of the stack.

The *lust in-first out* nature of this algorithm is written to complement the characteristics of short-term human memory. However, it is presented as an example only. The SemiAutomated procedure could be improved in several ways. For example, the command loop could include

- a command that lists the stack,
- a command that displays the search tree generated so far,
- a command that pops the stack until a particular state is on top (multiplepops),
- a command that performs `Analyze_Next_State` until a state is reached that either has no deviations, is inconsistent, or contains a hazardous deviation (multiple pushes).

6.2 Fully-Automated Analysis

The procedure Automated does a breadth-first search with respect to the derived states. First, all of the states with no assumptions are analyzed. Then the states with a single assumption are analyzed. The analysis continues until the maximum depth of assumptions (MAXDEPTH) is reached or until there are no more states to analyze.

The algorithm first obtains the initial state from the analyst. This state is appended to an empty queue. It is not first analyzed as in the semi-automated algorithm since the search loop performs this task.

The search loop first pulls a state `S` off the front of the queue. `S` is analyzed for definite deviations (discussed in detail in the next section.) In order for the

next state of S (Next(S)) and the S's derived states to be analyzed, S must satisfy certain criteria. First, it must contain at least one deviation. If it does not, then no states based on S will contain deviations. Second, it must be internally consistent—inconsistent states are not reachable according to the requirements and assumptions of the system. Third, it must be unique. If it is not unique, it means that this state has been reached via a different search path; any further analysis would be redundant. Fourth, if S contains a hazardous deviation then by definition the search is complete.

In addition, Automated prevents runaway searches by limiting the number of steps taken forward (MAX_STEPS) and the number of assumptions a search path can have (MAXDEPTH). The values for MAX-STEPS and MAXDEPTH depend on the characteristics of the system being investigated and should probably be determined by the analyst, although the algorithm could make suggestions based on the size of the causality diagram.

If S satisfies the criteria to continue forward, then S's successor is prepended to the queue. Thus, Next(S) is analyzed on the next iteration of the search loop.

If S satisfies the criteria to investigate potential derived states, then those states are generated by Propagate_Possible_Deviations (section 6.4) and appended to the queue. Since they are appended to the queue and the successors are prepended, this ensures that all of the successors are analyzed before the first derived state. When the last successor state is analyzed, the derived states are at the front of the queue. Their successors are prepended and as they are analyzed, the new, twice-derived states are appended. In this way, all of the states of one order of derivation are analyzed before any of a higher order of derivation.

Figure 6.3 shows an example of the search order. The boxes represent states in a search tree, where the initial state is the root of the tree (marked by the number

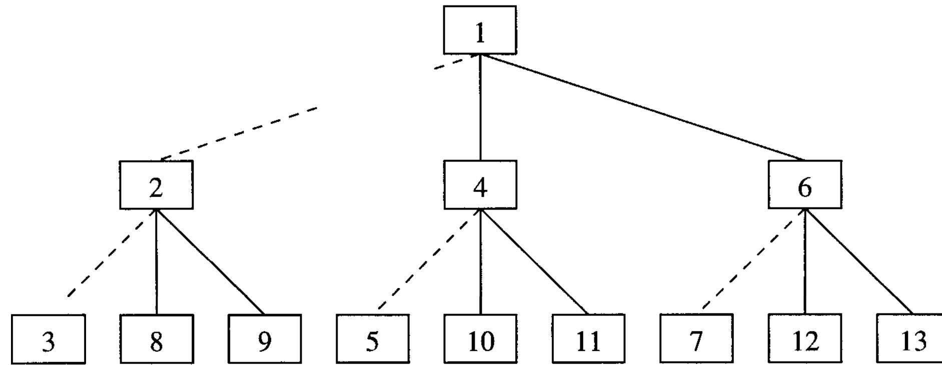


Figure **6.3:** An example of a search tree produced by the `Automated` procedure. The boxes represent states, the dashed lines mark successor states, the solid lines mark derived states, and the numbers show the order of analysis.

'1'). One of the children of each non-leaf state is its successor; they are connected by a dashed line. For the purpose of the example, each non-leaf state has two derived states — these states are connected to their base states by solid lines. The numbers in the boxes show the search order. Refer to table 6.1 for the execution trace corresponding to figure 6.3.

The algorithm given here is intended as an example of an efficient traversal of the search space. It is based on the premise that hazard scenarios requiring fewer assumptions are likelier than those with more assumptions. This premise is not necessarily true. Other search strategies that are more sensitive to the characteristics of the system being analyzed are possible. For example, the system variables could be prioritized according to likelihood to deviate. Thus, assumptions could be weighted. These weights could be used to sort the search queue. Such a strategy is outside the scope of this thesis, since it would likely require a detailed taxonomy of safety-critical systems.

State (S)	Operation	State Queue
	Analyst provides initial state	{1}
1	Pop front and analyze	{}
1	Prepend successor	{2}
1	Append derived states	{2,4,6}
2	Pop front and analyze	{4,6}
2	Prepend successor	{3,4,6}
2	Append derived states	{3,4,6,8,9}
3	Pop front and analyze	{4,6,8,9}
4	Pop front and analyze	{6,8,9}
4	Prepend successor	{5,6,8,9}
4	Append derived states	{5,6,8,9,10,11}
5	Pop front and analyze	{6,8,9,10,11}
6	Pop front and analyze	{8,9,10,11}
6	Prepend successor	{7,8,9,10,11}
6	Append derived states	{7,8,9,10,11,12,13}
7–13	Pop front and analyze	{}

Table 6.1: Execution trace of the Automated procedure. The left column is the state currently being analyzed. The middle column is the action being performed. The right column is the queue of states yet to be analyzed.

6.3 Propagation of Definite Deviations

The previous section presented the main loop for the Automated algorithm. This section describes the procedure PropagateDefiniteDeviations, shown in figure 6.4. This procedure takes a state as input and attempts to propagate any deviations it contains. In the process of doing this it also propagates any normal values that it can, since the normal values of some variables can condition whether and how deviations can occur in other variables.

Each state has two queues for the analysis of definite deviations. One queue is for the forward propagation of value changes. The other queue is for propagation backward, from effect-to-cause. The forward queue is appended under three circumstances:

```
procedure Propagate_Definite_Deviations(S : Mode)
  begin
    if ( Is_Derived_Mode(S) and not Finished(Base(S)) ) then
      Propagate-Definite_Deviations(Base(S)) ;
    Process_Definite_Queues(S);
    if ( Initial_Mode(S) or From_Base(S) ) then
      begin
        Process_Definite_Queues(Prev(S));
        Process-Def inite_Queues(S) ;
      end
    Set_Finished(S);
    Check-For-Equivalence@);
    Check_For_Hazards(S);
  end;

procedure Process_Definite_Queues(S : Mode)
  begin
    while not Empty(Backward_Queue(S)) do
      Propagate-Backward-Def inite(S, Pop-Front(Backward_Queue(S))) ;
    while not Empty(Forward_Queue(S)) do
      Propagate-Forward-Def inite(S, Pop-Front(Forward_Queue(S))) ;
  end;
```

Figure 6.4: The `PropagateDef initeDeviations` procedure.

- when an assumption about a node is added to a state (including the initial assumptions the analyst provides),
- when a node changes value due to a forward propagation (see discussion of `Propagate_Forward_Definite`), and
- when a node changes value due to a backward propagation (see discussion of PropagateBackwardDefinite).

The backward queue is appended in two circumstances:

- when an assumption about a node is added to a state (including the initial assumptions the analyst provides),
- when a node changes value due to a backward propagation (see discussion of PropagateBackwardDefinite).

The procedure takes a state S as input. If S is a derived state, then its base state must be analyzed before S is analyzed. Otherwise not all values will get propagated in S. For example, suppose node $N_1 = N_2$ A N_3. Also suppose that in order to propagate a deviation (unrelated to N_1), N_2 is assumed to be true in S. If N_3's value is unknown in Base(S), then N_1's value is also unknown, since

$$\text{true A } N_3 = N_3 = \text{unknown.}$$

If, however, some value in Base(S) propagates to cause N_3 to be set to true or false, then N_1 should be updated correspondingly in S. It will not, however, if N_1 is analyzed in S prior to N_3 in Base(S). Thus Base(S) should be analyzed before S (hence the first statement in the procedure.)

After analyzing the base state if necessary, S's definite queues are processed. The procedure `Process_Definite_Queues` first processes all of the nodes in the backward

definite queue (`Backward_Queue(S)` in the algorithm.) The backward definite queue is processed first because, as noted above, backward analysis can cause nodes to be added to both the backward queue and the forward queue, whereas the forward analysis can only result in nodes being added to the forward queue. Processing the backward queue results in an empty backward queue and a possibly larger forward queue. Processing the forward queue results in an empty forward queue and no change in the backward queue. Thus, processing backward and then forward results in two empty queues. Processing forward and then backward could result in a non-empty forward queue.

If `S` contains assumptions, then its previous state must also be analyzed because some values may propagate from `S` backward to its previous state and then back again.

For example, if node $N_1 = Previous(N_2) \vee N3,$ and an assumption of state S is that N_1 is false, then N_2 must be false in `Prev(S)` and N_3 false in `S`. If another node $N_4 = Previous(N_2) \wedge N_5$, then N_4 must be false in `S`. In order for these values to propagate properly, `S` must first be analyzed as described above. The backward propagations may add nodes to `Prev(S)`'s queues. Thus, `Prev(S)` must then be analyzed. The forward propagations in analyzing `Prev(S)` may add nodes to `S`'s forward queue. (It cannot result in adding any nodes to `S`'s backward queue because only the forward propagations in `Prev(S)` can affect `S`'s queues, and as previously discussed backward queues can only be added to by backward propagations.) Since `S`'s forward queue may now contain entries, it must be re-analyzed. The procedure simply calls `Process_Definite_Queues` again, although the check to see if the backward queue is non-empty is unnecessary.

Next, the algorithm sets a flag in `S` to signify that it has been analyzed for definite deviations. The flag is used to determine whether the base state needs to

be analyzed before analyzing a derived state, a possibly recursive procedure if the base state is itself a derived state. This situation is not possible for the Automated algorithm since it analyzes all base states before proceeding to their derived states (since the successor states are pushed onto the front of the queue and derived states are pushed onto the rear.) However, the SemiAutomated algorithm allows the analyst to analyze a state S's derived state before S's successor. The derived state's successor has S's successor as its base state. For example, in figure **6.3** state 2 is state 5's base state.

The state is then checked for equivalence with any other state in the search tree. Finally, it is checked for the existence of hazardous deviations. These algorithms as well as the definite propagations will be discussed presently.

6.3.1 Forward Propagation

Forward propagation is the application of a qualitative function to update a binding in a state. Each node in the causality diagram is associated with a qualitative function. If one of the inputs to that node has its value refined, then the value of the node may also need to be refined. Forward propagation may take one of three forms–structural, sequential, or combined–depending on the types of edges into the node.

The impetus for forward propagation is the update of a node's value. Since the update occurs for a particular state, the two arguments that must be passed to the `Propagate_Forward_Definite` procedure (figure 6.5) are a state S and a node `N`. Recall that the edges out of a node **`N`** lead to nodes with which `N` has a causal relationship. Accordingly, `Propagate_Forward_Definite` calculates the qualitative function

for each child node C by iterating over the edges out of N. If the edge is sequential, then the function must be evaluated in the context of Next(S). The variable Child_S is assigned to either S or Next(S), depending on the type of edge.

The variable Old-Value is assigned to the current binding of C in Child_S or *Unknown* if no binding exists. New_Binding is assigned to the value of C's qualitative function given the new values of its inputs. The function Input-Values supplies the function with values either from Child_S or Prev(Child_S), depending on the type of each edge into C.

If the new value is inconsistent with the old value, then Child_S is marked inconsistent. The topic of consistency will be discussed presently. Otherwise, if the node's value has been refined, then it is added to Child_S's forward queue, so that its new value may be propagated to its children. If C's value is not updated and N's value is a deviation, then C is added to the "possible" queue to see if any assumptions can be made to propagate the deviation.

The procedure finishes when all of the children of N have been inspected, thus concluding all possible ways that N's new value may be propagated forward (without making assumptions.)

6.3.2 Backward Propagation

Backward propagation is the application of the inverse of a qualitative function to update a state's binding. Its role in deviation analysis is to provide additional information about a state. Like forward propagation, backward propagation is initiated by the initial assumptions. However, since deviation analysis is primarily a forward search method, the Automated algorithm limits the use of backward propagation to

```
procedure Propagate_Forward_Definite(S : Mode, N : Node)
  var
    Child-S : Mode;
    C : Node;
  begin
    foreach arc A out of node N do
      begin
        if ( Sequential(A) ) then
          Child-S := Next(S);
        else
          Child-S := S;
        C := Child(A);
        Old-Value := Value(Child_S, C);
        New-Value := Forward_Mapping(Function(C),
                                     Input_Values(S, C));
        if ( Inconsistent(Old_Value, New-Value) ) then
          Mark_Inconsistent(Child_S);
        else
          begin
            if ( Old-Value <> New-Value ) then
              begin
                Update_Value(Child_S, C, Old-Value, New-Value);
                Add_To_Forward_Queue(Child_S, C);
              end;
            if ( Unknown_Deviation(New_Value) and
                 Deviant(Value(S, N)) ) then
              Add_To_Possible_Queue(Child_S, C);
          end;
      end;
  end;
```

Figure 6.5: The Propagate_Forward_Definite procedure.

the state containing the assumptions and possibly its preceding state (see sections 6.2 and **6.3.**) The SemiAutomated algorithm as written does not allow analyzing the previous state, but this could easily be added.[1]

6.3.3 Existence of Deviations

The existence of deviations is simple to track. When a state is created, a flag, say `Contains_Deviations`, is set to `false`. If a binding is added which contains a deviation, the flag is set to `true`.

6.3.4 Internal Consistency

When the new value is calculated, the old and new value might be inconsistent. Recalling that qualitative values are actually sets over the quantitative domain, the two values are inconsistent if the new value is not a refinement of the old value, *i.e.*, New-Value $\not\subseteq$ Old-Value. In other words, the new value must either be the same as the old value or describe a portion of the old value.

An inconsistent result may be due to inconsistent assumptions. Recalling the Intruder-Status diagram (figure 2.1), an example of two inconsistent assumptions is a state in which Descend and Other-Traffic are both active.

[1] To give the analyst the ability to propagate definite deviations backward in time, the following line could be added to the case statement:

```
Analyze_Previous_State:  Push_State(Prev(S));
```

```
procedure Propagate_Backward_Definite(S : Mode, N : Node)
  var
    Parent-S : Mode;
    P : Node;
  begin
    if ( Arity(N) = 0 ) then return;
    Old-Input-Values := Input_Values(S, N);
    New-Input-Values :=
      Backward_Mapping(Function(N), Old-Input-Values, Value(S, N));
    if (New-Input-Values = Inconsistent) then
      begin
        Mark-Inconsistent(S);
        return
      end;
    foreach parent arc A of node N do
      begin
        if ( Sequential(A) ) then
          Parent_S := Prev(S);
        else
          Parent-S := S;
        P := Parent(A) ;
        if (New_Input_Values(P) <> Old_Input_Values(P)) then
          begin
            Update_Value(Parent_S, Old_Input_Values(P),
                         New_Input_Values(P));
            Add_To_Forward_Queue(Parent_S, P);
            Add_To_Backward_Queue(Parent_S, P);
          end;
      end;
  end;
```

Figure 6.6: The PropagateBackwardDefinite procedure.

It is possible for inconsistencies to arise from an internally inconsistent calculus. For example, the following two qualitative axioms are inconsistent:

$$Positive \ x \ Unknown = Positive$$

$$Positive \ x \ Negative = Negative$$

The hypothesis "The product of a positive and negative is positive" is true according to the first axiom but false according to the second.

It is also possible for inconsistencies to arise indirectly from the requirements specification and thus from the causality diagram directly. This situation arises from loops in structural causality, in which a path may be traced from a node back to itself exclusively via structural edges. Such a definition constitutes a recursive, iterative function (such as a limit function or a fractal.) Although it may be useful in some cases to define an environmental variable recursively, analysis of the causality diagram for structural loops is probably prudent.[2] Note that loops involving at least one sequential edge do not present a problem, since a node's value is defined in terms of its value in a previous state.

6.3.5 Checking for Equivalence

It is desirable to identify equivalent states to avoid redundant searches. This is useful not only for an efficient automated algorithm, but to aid the analyst in recognizing patterns in the search space.

Checking for equivalence is a potentially costly operation. If the states contain an average of b bindings, and there are c states to check against, then a naive check

[2] An even more prudent approach is to analyze the original specification for structural loops.

for equivalence of a state with previously defined states is $O(cb^2)$, assuming the states are unsorted. If the states are sorted (an operation requiring time $O(b\log b)$), then the cost of a search for equivalence is $O(cb + b\log b)$.

The search for an equivalent state is conducted each time a new state is created. Neglecting the effect of duplicates on the number of states created and kept, the number of comparisons for n iteratively-created states is given by

$$\begin{aligned} & \sum_{i=1}^{n} ib + \text{blogb} \\ = & \frac{bn(n+1)}{2} + nb\log b \\ = & O(n^2 b + nb\log b) \end{aligned}$$

Thus the search for equivalent states can become quite costly as the size of the search and number of bindings increases.

An efficient method of comparing states is to compute a pseudo-random integer based on the bindings in a state. The integer can be used to exclude practically all states except those that are equivalent. This method is similar to a hash function. A hash function should distribute hash keys amongst the buckets as evenly as possible, to minimize the number of conflicts. Similarly, the proposed equivalence function seeks to distribute states evenly over a set of n-bit integers, which shall be called *equivalence codes.* Additionally, the distribution should be as random as possible, so that similar states do not have a significantly higher probability of having the same code. Thus, even though a search will very likely be concentrated around a group of similar states, their equivalence codes should be randomly distributed.

The code function takes a state as input, which is to say that it takes a list of bindings as input. An additional constraint on the code function is that it be commutative with respect to the order of the bindings. This avoids the need to sort

```
#include <stdlib.h>

unsigned Encode(unsigned x) { srand(x); return (unsigned) rand(); }

unsigned Equivalence_Code(Binding B)
{
  unsigned int code, num-bytes, i;
  unsigned char *bytes;

  code = 0;
  num-bytes = sizeof(B);
  bytes = (char*) &B;
  for (i = 0; i < num-bytes; i++)
    code = Encode( code ^ Encode( (unsigned) bytes[i]) );
  return code;
3
```

Figure 6.7: Sample C++ functions for calculating equivalence code.

the bindings so that they are always handed to the function in the same way. It also allows the equivalence code to be computed incrementally, as bindings are added. This is particularly useful when a derived state is created, as it can simply inherit the equivalence code of its base state.

For example, suppose a base state has the bindings b_1, b_2, and b_3 and its derived state has the binding b_4. Suppose also that another base state has the bindings b_1 and b_3 and its derived state has the bindings b_2 and b_4. Then the two derived states must have the same equivalence code, which should be different from the codes from the two base states.

The hash function is composed of two operators, one for translating a variable binding to a pseudo-random key and one for combining the keys into a single equivalence code. It is the combining function that must satisfy the commutative property. The translating function must satisfy the constraint that it yields the same value each time it is called for a particular binding (*i.e.*, node-value pair).

A suitably random translating function is the pseudo-random function contained in most programming language libraries. Figure 6.7 shows an example from the C programming language. When this example was run on 100,000 test cases (nodes numbered between 0 and 9,999 and values numbered between 0 and 9) there were three pairs of bindings with the same equivalence code.[3]

A suitably random hash code is integer addition of pseudo-randoms. Integer addition is addition modulo 2^N, where N is the number of bits. Thus, the distribution of the sum randomly distributed naturals between zero and $2^N - 1$ results is itself random.

6.3.6 Checking for Stopping Criteria

The stopping criterion can take one of several forms. It can be a list of nodes in which a deviation is considered hazardous. It can be a list of states that are considered hazardous. The former is faster to check for. The latter allows for node interaction, *e.g.*, one may interested in a state in which one node is too high while the other is too low.

6.4 Propagation of Possible Deviations

Conditional analysis is the propagation of possible deviations. These are deviations which, given certain assumptions about the state of the system over and above what is currently known about its state, will definitely occur.

[3]Experiment performed on a Mitra 486DX running Linux 1.1.2, using GNU gcc 2.5.8 and GNU stdlib. The `Binding` data structure was represented as `struct { unsigned short int Node; char Value;}`.

```
procedure Propagate_Possible_Deviations(S : Mode)
  begin
    while not Empty(Possible_Queue(S)) do
      Propagate_Possible(S, Pop_Front(Possible_Queue(S)));
  end;

procedure Propagate_Possible(S : Mode, N : Node)
  var
    D, Parent-D : Mode;
  begin
    if (Arity(N) == 0) then return;
    Old-Input-Values := Input_Values(S, N);
    Old-Value := Value(S, N);
    New-Values :=
      Possible_Mapping(Function(N), Old-Input-Values, Old-Value);
    if (New_Values(N) <> Old-Value) then
      begin
        D := Create_Derived_Mode(S);
        Update-Value(D, Old-Value, New-Values(N));
        Add-To-Forward-Queue(D, N);
        Add_To_Backward_Queue(D, N);
        for each parent arc A of N do
          begin
            if ( Sequential(A) ) then
              Parent-D := Prev(D);
            else
              Parent-D := D;
            P := Parent(A);
            if (New_Values(P) <> Old_Values(P)) then
              begin
                Update_Value(Parent_D, Old_Values(P),
                             New-Values(P));
                Add-To-Forward-Queue(Parent-D, P);
                Add_To_Backward_Queue(Parent_D, P);
              end;
          end;
      end;
  end;
```

Figure 6.8: The `PropagateBackwardDefinite` procedure.

Note that unlike the PropagateDefinite procedures, this procedure does not need to check whether the old and new values are inconsistent. This is because the definite procedures check for inconsistencies whenever a node changes value. If a value changes due to a forward propagation, then it is known to be consistent with respect to its inputs; it needs to be checked whether it is consistent with its children nodes. This will happen as soon as it is removed from the forward definite queue. Likewise a backward propagation causes a node's value to be consistent with one of its children nodes. Since backward propagations cause a changed node to be put into both the forward and backward queues, it will be checked against its inputs when it is analyzed by `Propagate_Backward_Definite` and against its other children nodes when it is analyzed by `Propagate_Forward_Definite`.

It is important to note that these assumptions may be inconsistent with the state of other system variables. Several things can cause this. Some assumptions about the system's behavior may have been left unspecified. Also, the qualitative mathematics presented in chapter 5 groups sets of values that are considered similar. This grouping facilitates analysis, but can result in loss of significant information. The result of impossible assumptions is that the analyst may be presented with an impossible hazard scenario. In this respect, deviation analysis is a conservative method similar to SMHA. The analyst must invest time in protecting against potentially impossible situations, but in doing so also provides insurance against false assumptions and future changes in the system's operating environment.

On the other hand, the assumptions made by the algorithm may actually be inevitable given the known state of the system. Again, this may be due to other unspecified assumptions made by the experts or due to the grouping effect of qualitative analysis. In this case, the analyst is presented with a scenario and told that a hazard *might* occur, when in point of fact it *will* occur. The analyst should still

take the scenario seriously, but a potential danger is that in the face of limited time and resources the analyst may give the hazard a lower priority than other less-likely hazards.

The forward possible queue is appended when a node's deviation does not propagate forward and one of the node's children has an unknown deviation value.[4] This indicates the possibility that assumptions could be made in order to propagate the deviation.

6.5 Summary

This chapter presented two related algorithms, one semi-autoamted and one automated, to assist the analyst in discovering potential problems in how software handles system deviations.

The author feels compelled to discuss two possible uses of deviation analysis that he feels are not advisable. Although the resulting trace of events from a forward search ties a hazard to a given set of causal factors, this information cannot be considered fulfillment of the goal of finding causal factors. The analyst cannot assume that all causal factors have been identified. Though it may be tempting to use the information to infer the impossibility of a hazardous event, this practice is not sound.

As discussed in chapter 1, the calculation of risk is a controversial task, due to the lack of confidence, statistical or otherwise, that can be placed in software failure probabilities. Although the results of the algorithm presented in this chapter can conceivably be used to calculate approximate risk given an initial scenario, a

[4] *N.B.:* "Unknown" means that a value may or may not be deviant from the norm, not that it is definitely a deviation of unknown magnitude.

full forward search, and probabilities on the assumed events, such an analysis would require a careful treatment of variable interaction, since a wrong assumption that system variables are independent can lead to failure probabilities that are many orders of magnitude too optimistic. Therefore, the author advises against such an analysis.

Chapter 7

Examples

This chapter presents experimental results of the deviation analysis algorithm and a discussion of how deviation analysis addresses the goals of this dissertation. The experiments were performed on two models. One model is a simple train crossing example and the other is part of the requirements specification for the TCAS avionics software. The results for each experiment will take the form of space and time requirements and some example scenarios produced by the algorithm.

The software used in these experiments utilizes the RSML simulator developed at the University of California, Irvine and the University of Washington. The majority of the deviation analysis software is written in C++. The top-level search algorithms presented in chapter 6 are written in the interpreted language Tcl. The compilation and experiments were performed on the Linux 32-bit operating system (version 1.1.2) using the GNU C++ compiler (version 2.5.8). Size and execution performance data are for unoptimized code (including debugging information) on an Intel 80486DX2 microprocessor at 66 MHz. Available physical memory ranged from six to twelve megabytes before loading the code and data. The initial free memory affects how soon before virtual memory swaps begin, so execution figures are only approximately related. Sixteen megabytes of virtual memory were available for the experiments.

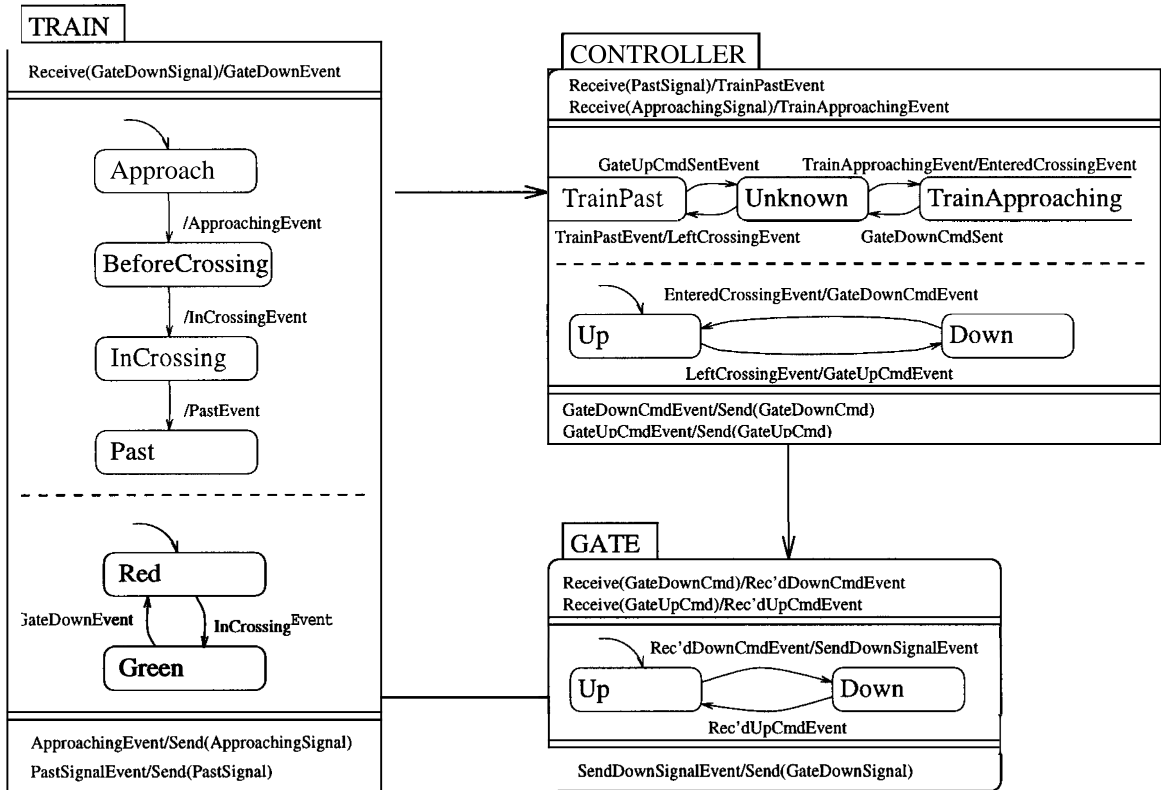


Figure 7.1: Train crossing example.

The parameters for $P_{B,N}$ are as follows. The base (B) equals 4 and the number of elements on each side of zero (N) is 8. Thus, the domain has the following divisions:

$$(-16384, -4096, -1024, -256, -64, -16, -4, 0, 4, 16, 64, 256, 1024, 4096, 16384\}$$

The major consideration in choosing the parameters was to make sure that the divisions reached into the tens of thousands, since the altitude variables in the avionics example can reach 30,000 feet or more. N was chosen to be large enough to provide roughly order-of-magnitude coverage within B^N.

This chapter introduces some terminology. Deviations of nodes that have been identified by the analyst as being significant are referred to as "significant deviations." "Significant states" are states that contain significant deviations. A trace from the initial state to a significant state is a "scenario."

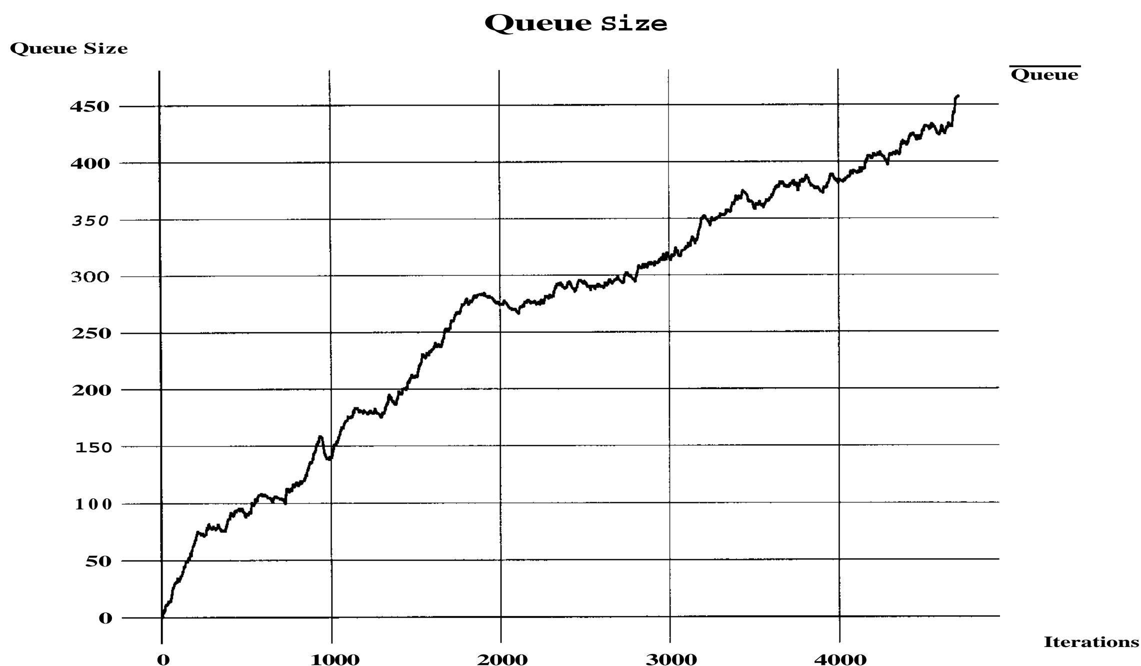


Figure 7.2: Queue sizes on train crossing example 1.

7.1 Train Crossing

A specification for a simple system is shown in figure 7.1. As the train approaches a railroad crossing, a sensor on the tracks sends a signal to the gate controller, which then lowers the gate. After the train has left the intersection, another sensor signals the controller to raise the crossing gate.

The software creates the causality diagram in about 1.5 seconds. The data structures require 252 kilobytes. There are a total of 980 nodes in the causality diagram, with 65 nodes directly mapped to entities in the RSML specification and the balance of the nodes encoding the causal relationships of the somewhat rich RSML semantics.

The analyst makes initial assumptions — including at least one deviation — to start the analysis. In this case, one might investigate the potential effects of the

```
ASSUMPTIONS:
Step 0: node557: dev(ApproachingSignalEvent) = T
Step 0: node50: value(ApproachingSignalEvent) = F
Step 2: node911: correct(value(TrainPositionUnknown)) = T
Step 3: node972: correct(value(GateUp)) = T
VALUES:
Step 0: node50: value(ApproachingEvent) = F
Step 1: node68: value(SendApproachingSignal) = F
Step 2: node183: value(Unknown-to-TrainApproaching) = F
Step 2: node69: value(ReceiveApproachingSignal TrainApproachingEvent) = F
Step 3: node156: value(EnteredCrossingEvent) = F
Step 3: node213: value(GateUp-to-GateDown) = F
Step 4: node190: value(TrainApproaching-to-Unknown) = F
Step 4: node150: value(GateDownCmdEvent GateDownCmdSentEvent) = F
Step 5: node171: value(SendGateDownCmd) = F
DEVIATIONS:
Step 0: node557: dev(ApproachingEvent) = T
Step 1: node589: dev(SendApproachingSignal) = T
Step 2: node714: dev(Unknown-to-TrainApproaching) = T
Step 2: node874: dev(ReceiveApproachingSignal TrainApproachingEvent) = T
Step 3: node832: dev(GateUp-to-GateDown) = T
Step 3: node871: dev(EnteredCrossingEvent) = T
Step 4: node869: dev(GateDownCmdEvent GateDownCmdSentEvent) = T
Step 5: node898: dev(SendGateDownCmd) = T
```

Table 7.1: Summarized results of one of the train crossing's scenarios.

approach signal not being sent to the controller when it should have. The analyst must also select significant nodes. For the first example, the only significant node is the gate-down command issued by the controller. With only the one significant node, the search grows very large. Figure 7.2 is a graph of the state queue size for each iteration of the `Automated` algorithm. `Automated's` main loop made 4,700 iterations before exhausting available memory on the computer (approximately 24 Mbytes.) The angle of the curve suggests that the algorithm is not yet halfway through the search.

Although the search did not terminate, it still yielded useful information. Recall that deviation analysis is meant as an investigative method, not an exhaustive analysis. In fact, the most general results are the first to be generated since the algorithm

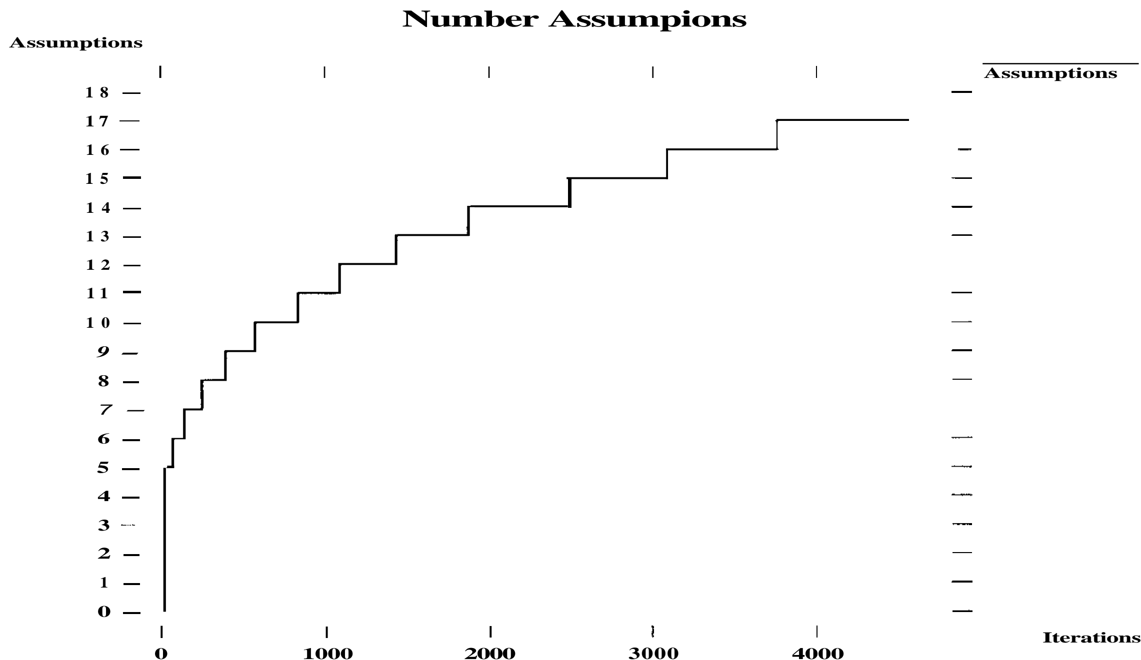


Figure 7.3: (Train crossing, example 1.) The x axis is the number of iterations of **Automated.** The y axis is the number of assumptions made to arrive at the state being inspected in iteration i.

investigates all scenarios with n assumptions before moving on to $n + 1$. Table 7.1 shows the first significant state to be returned by the algorithm. The scenario requires four total assumptions. The first two assumptions are the ones provided by the analyst: the signal for the approaching train does not occur when it should. The next assumption occurs two steps later—the controller's model of the train should be in state *Unknown.'* Finally, the controller model of the gate should be in state *GateUp* in the following step. Given these assumptions, the algorithm determines the values and deviations shown. In particular, it finds that under these circumstances, the controller should send the command to lower the crossing gate but does not.

Chapter 6 asserted that the search space can be pruned by limiting the number of steps or assumptions. Given that the number of assumptions increases monotonically as the search progresses, a limit on the number of searches amounts to a stopping

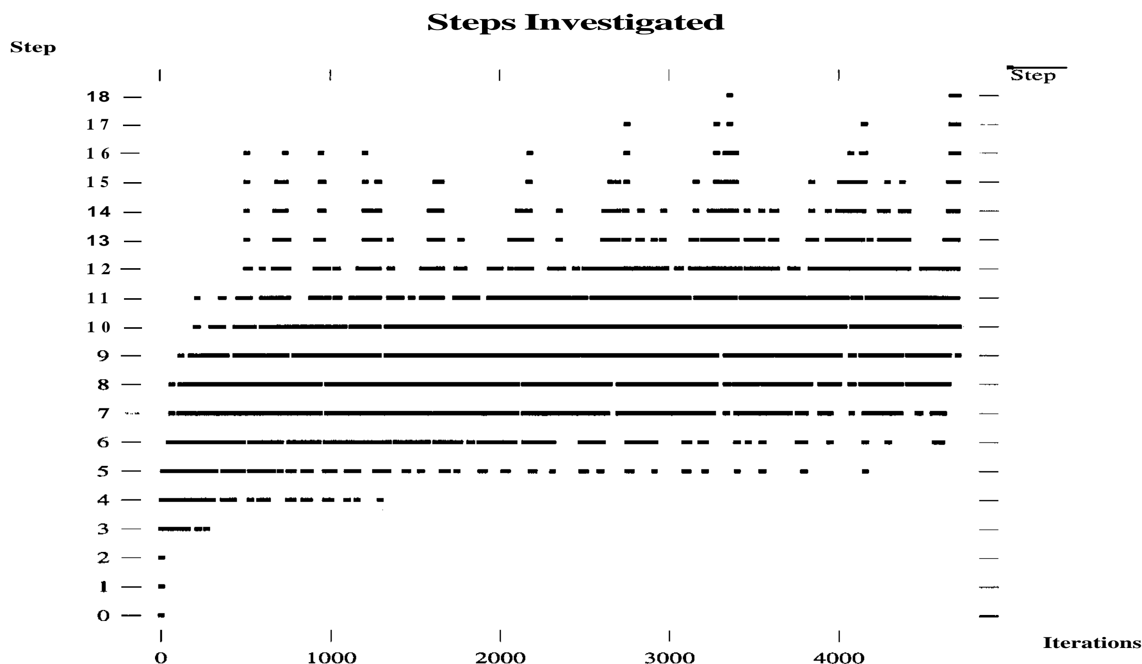


Figure 7.4: (Train crossing, example 1.) Number of steps taken after initial assumption to arrive at the state under investigation at each iteration.

criterion for the algorithm (see figure 7.3.) Limiting the number of steps can prune out much of the search space if the analyst is interested in immediate effects. Figure 7.4 shows the number of steps (sequential arcs followed) to get to the state at each iteration i. Figures 7.5 and 7.6 shows the queue and assumption graphs when the maximum number of steps is limited to five (example 2). Note that the search was not limited as to the number of assumptions. The algorithm terminated after **98** seconds and **383** iterations, finding four significant states. The search required 2.6 megabytes of memory.

The search may also be pruned by adding significant nodes. The first example showed that the search space grows very large when targeting a single significant node. Figure 7.7 compares the first example with a third search on both of the controller's

'Recall from chapter 5 that "value(*Unknown*) $\oplus$ dev(*Unknown*)" represents the correct value of *Unknown*. The software converts this expression to "correct(*Unknown*)" for display to the analyst.

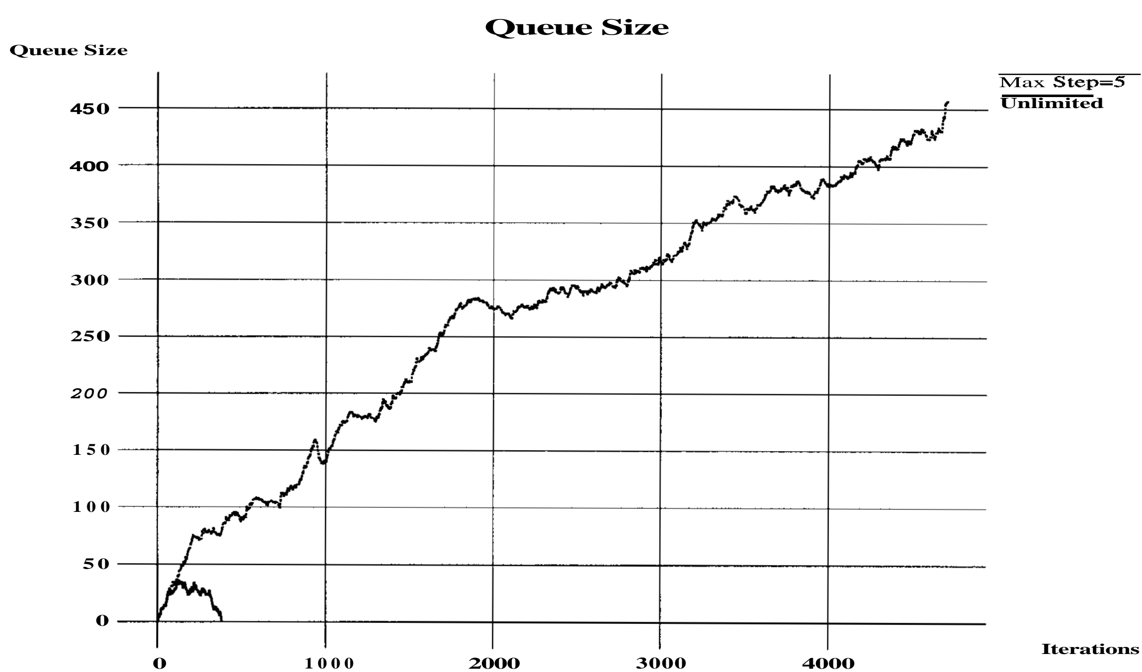


Figure 7.5: Search queue comparison for train crossing examples 1 and 2.

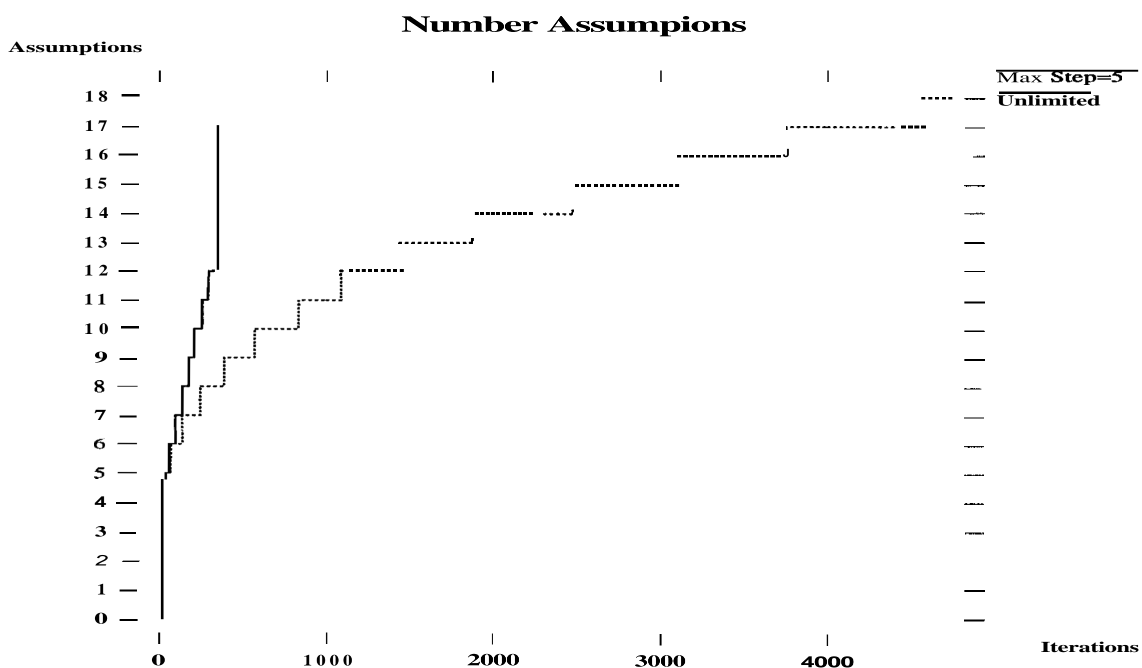


Figure 7.6: Assumption comparison for train crossing examples 1 and 2.

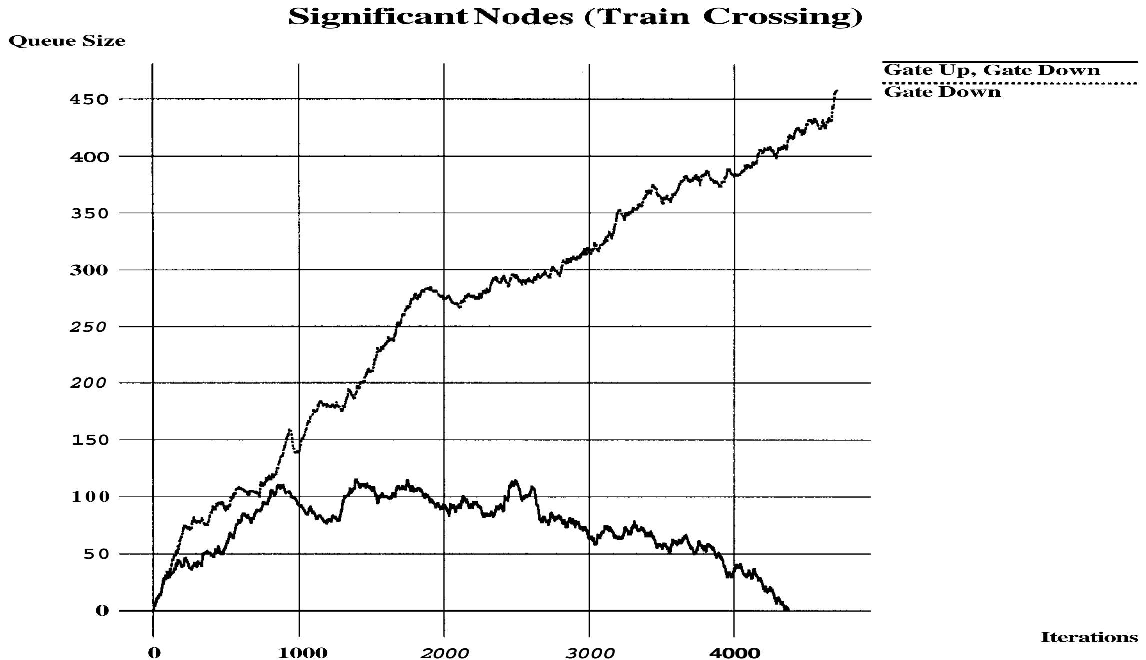


Figure 7.7: Search queue comparison for train crossing examples 1 and 3.

outputs (gate-up and gate-down commands.) The third example terminated after 24.2 minutes. The search required 21.3 Mbytes of memory to iterate over 4,373 states. A total of 220 significant states were discovered.

7.2 TCAS II

Chapter 3 introduced the TCAS II avionics system. The portion of the specification that deals with the model of own aircraft (*i.e.*, the aircraft containing the TCAS unit) will serve as an example. Figure 7.8 shows an RSML diagram of the states in the *Own-Aircraft* model. The top part of the diagram lists inputs from other components, the states in the middle represent knowledge about the state of own aircraft (including commands received from TCAS), and the bottom of the diagram lists outputs to other aircraft components, such as the TCAS display and the

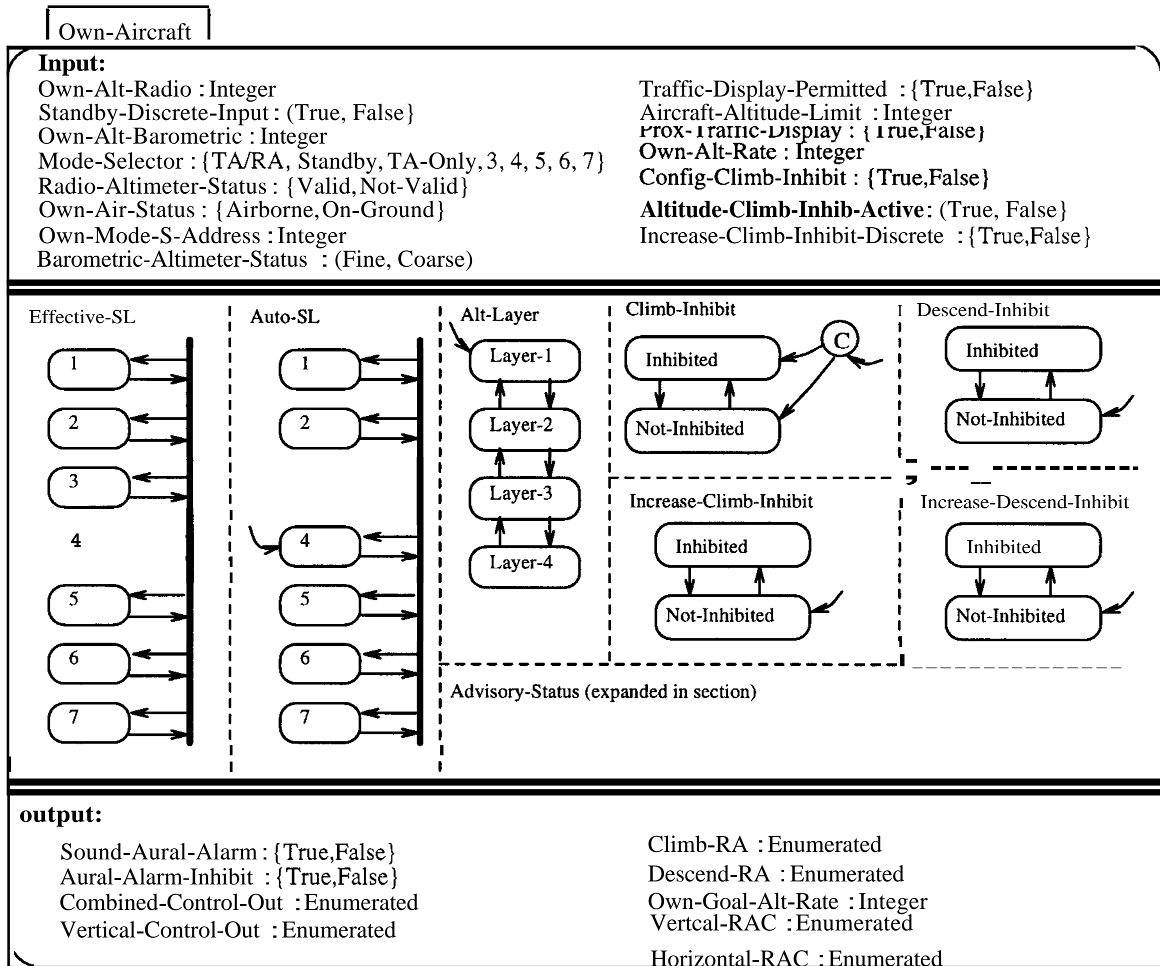


Figure 7.8: The TCAS II model of its own aircraft.

transmitter. The *Advisory-Status* and transition logic (which is quite substantial) are not shown in the figure. The relevant parts of this diagram will be explained as needed.

The *Own-Aircraft* causality diagram contains 322 nodes directly related to items of the specification (state names, event names, transition names, *etc.)* and 9675 total nodes. The causality diagram for *Own-Aircraft* requires 2.8megabytes of memory. The data structure is created in approximately 11 seconds.

One of the inputs to the *Own-Aircraft* model is the altitude above sea-level, provided by the aircraft's barometric altimeter *(Own-Alt-Barometric* in the figure.)

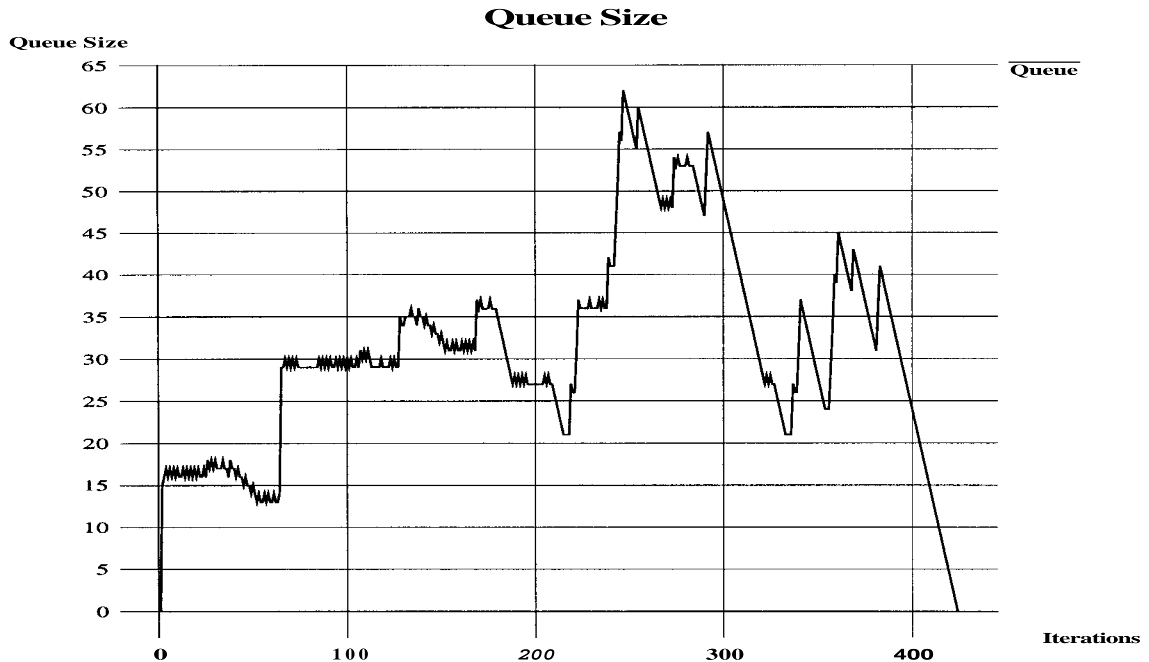


Figure 7.9: Queue sizes on TCAS II example.

Suppose that the analyst wishes to know potential effects of the barometric altimeter reading being too high. In the example the deviation of *Own-Alt-Barometric* is assumed to take the qualitative value 1, 2, or 3, which corresponds to

$$0 < \text{dev}(\textit{Own-Alt-Barometric}) \leq \mathbf{64}.$$

The analyst next identifies the "significant" nodes, *i.e.*, the nodes for which deviations are considered to be of interest. In this example, assume that any transition is considered to be significant. In other words, if a transition is taken when it should not be, or is not taken when it should, then report the assumptions that led to the deviation.

Figure 7.9 is a graph of search queue size versus iteration. The search queue contains the intial state before the first iteration and is empty again after 424 iterations. The queue contains a maximum of 62 states. The entire search took 23.8

minutes. 1,473 total states were created. The total memory used in the search was 3.2 Mbytes.

The algorithm identified 144 significant states. All of the significant states ended with deviations of automatic sensitivity level (ASL). ASL is a measure of TCAS's sensitivity, which can be viewed as a protective sphere around the aircraft. The ASL can range between a value of 1 and 7. The higher the ASL, the larger the protective sphere. ASL 1 affords no sphere of protection and displays no traffic information. While in ASL 7, TCAS attempts to give a warning 35 seconds prior to the point of closest approach with a threatening aircraft. The inputs to ASL are the altimeter readings and the prior "effective" sensitivity level (ESL), which is based on pilot and air traffic control input.

One scenario output by the program involves a failure to transition from ASL 1 to 5. Recall that ASL 1 provides no protection. On the other hand, ASL 5 provides a warning of 25 seconds. If ASL fails to transition from 1 to 5, then TCAS is not providing the protection that it should be. The output for the scenario is listed in table 7.2. In the lists of values and deviations there is a combination of `dev(ASL_1-to-ASL_5) = T` and `value(ASL_1-to-ASL_5) = F`. Together they mean that the transition should have occurred but did not.

The first assumption in the list is the analyst's assumption. The remaining assumptions are made by Automated in an attempt to propagate the deviation through the *ASL_1* to *ASL_5* transition, shown in figure 7.10. The barometric altitude reading appears twice in the table, in the third and fourth rows.[2] A deviation in the fourth row is what occurs in this scenario, and that is the meaning of the second assumption.

[2]The deviation for the third row eventually leads to a transition from **ASL** 1 to 5 that should not occur but does, enabling TCAS alarms when they should be disabled.

Transition(s): □ ⟶ □

Location: Own-Aircraft ▹ Auto-SL$_{s-30}$
Trigger Event: Descend-Inhibit-Evaluated-Event$_{e-279}$
Condition:

AND		OR	
Effective-SL$_{s-30}$ **in state 4**	T	T	·
Effective-SL$_{s-30}$ **in state 6**	·	·	T
Own-Alt-Barometric$_{v-33}$ $\geq$2550 ft$_{(ZSL4TO5)}$	·	T	·
Own-Alt-Barometric$_{v-33}$ $\leq$9500 ft$_{(ZSL6TO5)}$	·	·	T
Own-Alt-Radio$_{v-31}$ $\geq$2550 ft$_{(ZSL4TO5)}$	T	·	·
Climb-Desc.-Inhibit$_{m-198}$	F	F	F
Own-Air-Status$_{v-36}$ = Airborne	T	T	T
Radio-Altimeter-Status$_{v-35}$ = Valid	T	F	·

Output Action: Auto-SL-Evaluated-Event$_{e-279}$

Figure 7.10: The automatic sensitivity level transition from state 1 to state 5.

In order for the high value to propagate, the value of *Own-Alt-Barometric* must be greater than the 9500-foot threshold (*ZSL6TO5*) and the amount of the deviation must be greater than the amount it is above that threshold. In other words, without the deviation, the altimeter reading would have been less than or equal to 9500 feet.

Now that the deviation has propagated to the fourth row, assumptions must be made in order to propagate it to the entire column. Since the fourth row should have been true but is not, the other three relevant rows should be satisified so that the column's value is dependent on the fourth row, and hence the deviation. The third, fourth, and fifth assumptions address these rows. Note that the assumptions are not on the actual values but the *correct* values. This is a more general assumption, since a deviation in any of these other rows also propagates a deviation. The analyst may wish to assume that the actual values are correct (and in fact the `Automated` algorithm may assume this at a later point of the search) but at this point it is sufficient to assume that the effective sensitivity level should be in state 6, the aircraft should not be inhibited from climbing and descending, and the aircraft should be airborne.

```
STATE: state12241   STATUS: has-deviations significant from-base
# STEPS: 0   # ADDED ASSUMPTIONS: 7
ASSUMPTIONS:
Step 0: node4949: dev(Own_Alt_Barometric) == 1 2 3
Step 0: node5729: ZSL6T05 < value(Own_Alt_Barometric) AND
                  ZSL6T05 - value(Own_Alt_Barometric) >
                                              -dev(Own_Alt_Barometric)
Step 0: node5759: correct(value(ESL_6 Effective_SL In State ESL_6))
Step 0: node5769: correct(NOT value(Climb_Desc_Inhibit))
Step 0: node5750: correct(value(Own_Air_Status == Airborne))
Step 0: node5827: NOT value(ESL_4 Effective-SL In State ESL_4) OR
                  value(Climb_Desc_Inhibit) OR
                  NOT value(Own_Air_Status == Airborne) OR
                  value(Radio_Altimeter_Status == Valid)
Step 0: node5705: NOT value(ESL_4 Effective-SL In State ESL_4) OR
                  NOT value(Own_Alt_Radio >= ZSL4T05) OR
                  value(Climb_Desc_Inhibit) OR
                  NOT value(Own_Air_Status == Airborne) OR
                  NOT value(Radio_Altimeter_Status == Valid)
Step 0: node5688: correct(value(ASL_1 Auto-SL In State ASL_1) AND
                  value(Descend_Inhibit_Evaluated_Event))
VALUES:
Step 0: node1072: value(ASL_6-to-ASL_5) = F
Step 0: node780: value(ASL_4-to-ASL_5) = F
Step 0: node634: value(ASL_2-to-ASL_5) = F
Step 0: node1218: value(ASL_7-to-ASL_5) = F
Step 0: node926: value(ASL_5-to-ASL_5) = F
Step 0: node468: value(ASL_1-to-ASL_5) = F
Step 0: node291: value(Own_Alt_Barometric) = 7 8
Step 0: node474: value(Own_Alt_Barometric >= ZSL4T05) = T
Step 0: node478: value(Own_Alt_Barometric <= ZSL6T05) = F
DEVIATIONS:
Step 0: node5692: dev(ASL_1-to-ASL_5) = T
Step 0: node5731: dev(Own_Alt_Barometric <= ZSL6T05) = T
Step 0: node4949: dev(Own_Alt_Barometric) = 1 2 3
```

Table 7.2: Scenario produced for TCAS II barometric altimeter deviation.

Recall from chapter 3 that an AND/OR table is true if one of its columns is true. The third column is false when it should be true, but this deviation does not propagate if one of the other columns is true. The sixth assumption addresses the second column of the AND/OR table. The expression is the negation of the second column, except that the third column cannot be false, since *Own-Alt-Barometric* > 9500 feet. For the second column the actual rather than correct values are assumed because a deviation in the second column may cancel out the barometric altimeter deviation. For example, if the assumption were correct (NOT `ESL_4`),[3] then ESL would be allowed to be in state 4, and if the other conditions are also satisfied the AND/OR table evaluates to true despite the deviation in the barometric altimeter. Thus, the deviation does not propagate.

The seventh assumption causes the first column to be false. Note that both the first and second columns can be made false by not being in ESL 4, being climb-descend inhibited, or not being airborne. Interestingly, the latter two rows cause the third column also to be false. Combining one of these with the the third, fourth, and fifth assumptions implies multiple, possibly independent, deviations. A strength of deviation analysis is that its search is not limited to single failures. Scenarios such as this will hopefully help the analyst to consider novel and potentially complex interactions between system variables.

The final assumption involves propagating the AND/OR table to the transition. ASL should be in state 1 and the trigger event should occur in order for the deviation

[3]The reader may observe that the expression should actually be in the form `NOT correct(X)`, but $(\neg X)_c = (\neg X)_a \oplus (\neg X)_d = \neg X_a \oplus X_d = \neg(X_a \oplus X_d) = \neg X_c$. It also makes intuitive sense that "It should be true that X is not true" is the same as "It is not true that X should be true," since boolean algebra conforms to the law of excluded middles (a value that is not true must be false.)

to propagate. Generally the analyst may assume that they actually are true, but like the third column assumptions, the possibility of multiple deviations is left open.

A trace of the `Automated` algorithm is listed in table 7.3. A graphical representation of this trace is provided by figure 7.11. The nodes in the graph represent nodes in the causality diagram. (Nodes 5753, 5827, and 491 appear twice to fit the graph on a page; each pair is a single node.) Edges are causal relationships, with the parents below the children. For example, node 5753 (in the top-lefthand corner) is dependent on node 5751 and node 5752 for its value. Arrows represent the direction of the search. Arrows pointing toward the top indicate that a parent node changed a child node's value via *Propagate_Forward_Definite*. A downward arrow indicates that ***Propagate-Backward-Definite*** changed a parent's value based on the child's value. The nodes' function and value are given except where irrelevent to propagation. Although space does not allow a full treatment of the meaning of each and every node, the intuitive meaning of certain nodes is given in table 7.6. The reader may also refer to table 7.2.

7.3 Conclusions

The experiments show that deviation analysis can take a formal specification and given one or more initial assumptions and one or more significant nodes can produce scenarios that are likely to be of interest to the safety analyst. The inputs to and output from the algorithm can be expressed in terms of the source specification, making it straightforward to use.

Interestingly, increasing the number of "significant nodes" increases the generality of the search, but actually constrains it in that the search size can only be reduced

```
Init:
        state_1: Created as initial state.
        state_1: Assume node4949 is {1 2 3}.
state_1: Forward Definite: Considering node4949:
        state_1: node5723 becomes F due to node4949's new value.
        state_1: node5724 becomes T due to node4949's new value.
state_1: Forward Definite: Considering node5723:
        state_1: node5728 becomes F due to node5723's new value.
state_1: Forward Definite: Considering node5724:
        state_1: Add node5730 to Forward Possible queue.
state_1: Forward Possible: Considering node5730:
        state_2: Created as derived of state_1 to propagate
                 node5730 as T.
        state_2: Assume node5729 is T.
state_2: Forward Definite: Considering node5730:
        state_2: node5731 becomes T due to node5730's new value.
state_2: Forward Definite: Considering node5731:
        state_2: Add node5761 to Forward Possible queue.
state_2: Backward Definite: Considering node5729:
        state_2: node5721 becomes T due to node5729's new value.
        state_2: node5725 becomes T due to node5729's new value.
state_2: Forward Definite: Considering node5721:
        state_2: node478 becomes F due to node5721's new value.
state_2: Forward Definite: Considering node478:
        state_2: node490 becomes F due to node478's new value.
state_2: Forward Definite: Considering node490:
        state11838: node491 becomes F due to node490's new value.
state_2: Forward Definite: Considering node491:
        state_2: node488 becomes F due to node491's new value.
state_2: Forward Definite: Considering node488:
        state_2: node5818 becomes T due to node488's new value.
        state_2: node5825 becomes F due to node488's new value.
state_2: Forward Possible: Considering node5761:
        state_3: Created as derived of state_2 to propagate
                 node5761 as T.
        state_3: Assume node5759 is T.
```

Table **7.3:** Execution order of the Automated algorithm on the TCAS II example (continued on next page)

```
state_3: Forward Definite: Considering node5761:
        state_3: node5763 becomes T due to node5761's new value.
state_3: Forward Definite: Considering node5763:
        state_3: Add node5771 to Forward Possible queue.
state_3: Forward Possible: Considering node5771:
        state_4: Created as derived of state_3 to propagate
                 node5771 as T.
        state_4: Assume node5769 is T.
state_4: Forward Definite: Considering node5771:
        state_4: node5744 becomes T due to node5771's new value.
state_4: Forward Definite: Considering node5744:
        state_4: Add node5752 to Forward Possible queue.
state_4: Forward Possible: Considering node5752:
        state_5: Created as derived of state_4 to propagate
                 node5752 as T.
        state_5: Assume node5750 is T.
state_5: Forward Definite: Considering node5752:
        state_5: node5753 becomes T due to node5752's new value.
state_5: Forward Definite: Considering node5753:
        state_5: Add node5829 to Forward Possible queue.
state_5: Forward Possible: Considering node5829:
        state_6: Created as derived of state_5 to propagate
                 node5829 as T.
        state_6: Assume node5827 is T.
state_6: Forward Definite: Considering node5829:
        state_6: node5693 becomes T due to node5829's new value.
state_6: Forward Definite: Considering node5693:
        state_6: Add node5707 to Forward Possible queue.
state_6: Backward Definite: Considering node5827:
        state_6: node5819 becomes T due to node5827's new value.
state_6: Backward Definite: Considering node5819:
        state_6: node5817 becomes T due to node5819's new value.
state_6: Backward Definite: Considering node5817:
        state_6: node492 becomes F due to node5817's new value.
state_6: Forward Definite: Considering node492:
        state_6: node498 becomes F due to node492's new value.
```

Table 7.4: Execution order of the Automated algorithm on the TCAS II example (continued on next page)

```
state-6: Forward Definite: Considering node498:
         state-6: node5696 becomes T due to node498's new value.
         state-6: node5703 becomes F due to node498's new value.
state-6: Forward Possible: Considering node5707:
         state-7: Created as derived of state-6 to propagate
                  node5707 as T.
         state-7: Assume node5705 is T.
state-7: Forward Definite: Considering node5707:
         state-7: node5683 becomes T due to node5707's new value.
state-7: Forward Definite: Considering node5683:
         state-7: Add node5690 to Forward Possible queue.
state-7: Backward Definite: Considering node5705:
         state-7: node5697 becomes T due to node5705's new value.
state-7: Backward Definite: Considering node5697:
         state-7: node5695 becomes T due to node5697's new value.
state-7: Backward Definite: Considering node5695:
         state-7: node497 becomes F due to node5695's new value.
state-7: Forward Definite: Considering node497:
         state-7: node469 becomes F due to node497's new value.
state-7: Forward Definite: Considering node469:
         state-7: node468 becomes F due to node469's new value.
state-7: Forward Possible: Considering node5690:
         state-8: Created as derived of state-7 to propagate
                  node5690 as T.
         state-8: Assume node5688 is T.
state-8: Forward Definite: Considering node5690:
         state-8: node5692 becomes T due to node5690's new value.
```

Table 7.5: Execution order of the Automated algorithm (continued.)

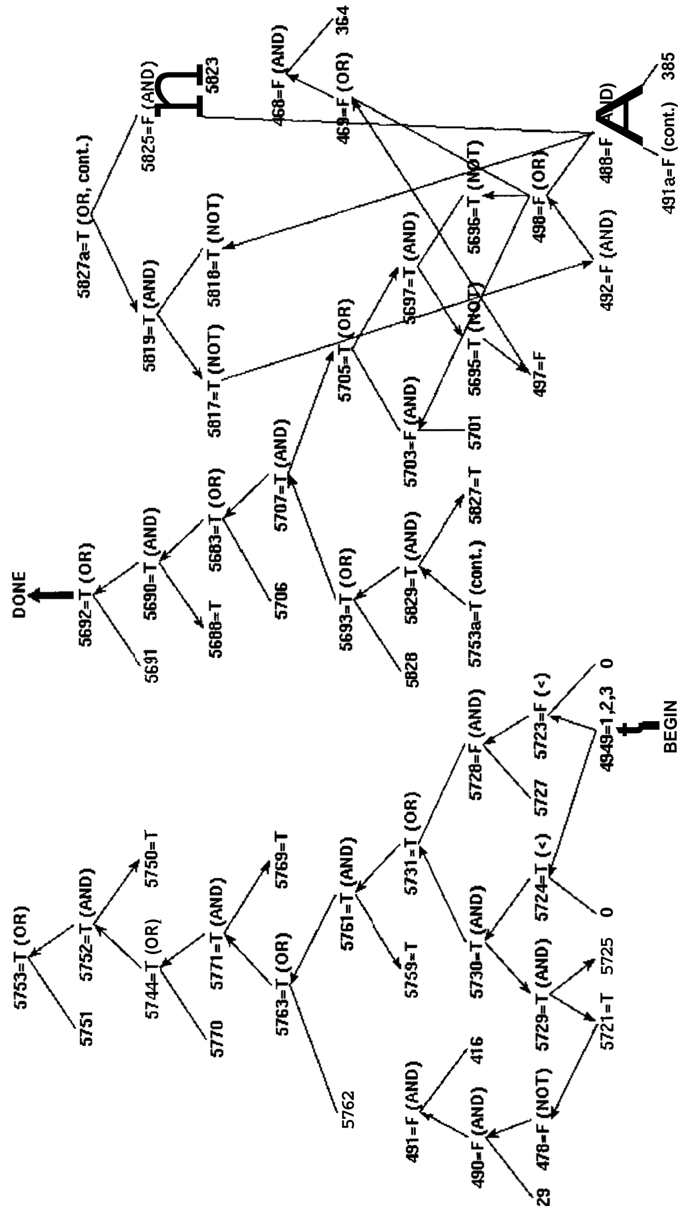


Figure 7.11: Search order of `Automated` on the TCAS II example.

Node	Meaning
29	ESL in state 6
385	air status is airborne (row 7)
416	not climb or descend inhibited
468	ASL in state 1 and triggering event is present
468	Transition ASL 1 to 5
469	AND/OR table (guarding condition for the transition) is satisfied
478	actual value of barometric altimeter $\leq 9500 feet$.
488	Column **3** value
490	Rows 2 and 4 in AND/OR table are true
491	Rows 2 and 4 are true and row 6 is false
492	Column 2 value
497	Column 1 value
498	Column 2 or 3 is satisfied
4949	barometric altimeter deviation
5683	AND/OR table deviation
5688	Conditions under which table deviation propagates to transition
5690	transition deviation, due to table deviation
5691	Transition deviation, due to deviation in ASL 1 or trigger event
5692	Transition ASL 1 to 5 deviation
5693	Deviation in "column 2 OR column 3"
5695	Column 1 is not satisfied
5696	Neither column 2 nor 3 is satisfied
5697	All **3** columns are false (AND/OR table is false)
5701	Column 1 should be false
5703	column 1 should be false, columns 2 and 3 are false
5705	Conditions under which col. 2 and 3 deviation propagates to whole table
5706	AND/OR table deviation, due to column 1
5707	AND/OR table deviation, due to "column 2 OR columnn 3"
5721	actual value of barometric altimeter > 9500 feet.
5723	barometric altimeter reading too low
5724	barometric altimeter reading too high

Table 7.6: Informal meaning of causality diagram nodes in the TCAS II example (continued on next page.)

Node	Meaning
5725	deviation is greater than difference between correct and actual values
5727	Conditions under which "*Own-Alt-Barometric* $\leq$ 9500 feet" propagates.
5728	barometric altimeter $\leq$ 9500 feet and it shouldn't be
5729	Amount that `Own-Alt-Barometric` is more than 9500 feet is less than deviation
5730	`Own-Alt-Barometric` > 9500 feet and it shouldn't be
5731	`Own-Alt-Barometric` $\leq$ 9500 ft is a deviation
5744	Column 3: rows 2, 4, and 6 are deviation
5750	Conditions to propagate deviation in rows 2, 4, and 6 to all of column 3
5751	Column 3 is a deviation due to row 7
5752	Column 3 is a deviation due to rows 2, 4, and 6
5753	Column 3 deviation
5759	Column 3: conditions to propagate deviation in row 2 to rows 2 and 4
5761	Column 3: rows 2 and 4 are deviation because of row 4
5762	Column 3: rows 2 and 4 are deviation because of row 2
5763	Column 3: rows 2 and 4 of the AND/OR table deviation
5769	Column 3: conditions to propagate row 2 or 4 to rows 2, 4, and 6
5770	Column 3: rows 2, 4, and 6 are deviation due to row 6
5771	Column 3: rows 2, 4, and 6 are deviation due to row 2 or 4
5817	Column 2 is false
5818	Column 3 is false
5819	Columns 2 and 3 are false
5823	Column 2 should be false
5825	Column 3 is true and column 2 should be false
5827	Conditions under which col. **3** deviation propagates to col. 2 and **3**
5828	Column 2 deviation propagates to columns 2 and 3
5829	Column 3 deviation propagates to columns 2 and 3

Table 7.7: Informal meaning of nodes in the TCAS II example (continued.)

by adding significant nodes. As expected, the **Automated** algorithm appears to be at a disadvantage when investigating effects on a single significant node. **A** directed investigation stilll may best be performed via a backward search, such as a fault tree analysis. Deviation analysis appears to be most useful when it is open-ended (many critical nodes), investigating the effects of a small number of assumptions on a larger number of significant nodes.

The software requires a large amount of memory (sometimes in the tens of megabytes) but the execution time is under **30** minutes for all experiments performed. Even when the algorithm exhausted memory, it was after it had identified many scenarios.

Chapter 8

Results and Future Directions

8.1 Results

This dissertation presented a new forward analysis method for safety-critical software. The method utilizes a primitive language of causality, a calculus of deviations, and either a semi-automated or automated search procedure.

A Primitive Language of Causality. The primitive language of causality was shown to be powerful enough to encode state-based specifications using RSML as an example. The language lends itself well to analysis since tracing causal relationships is reduced to traversing the diagram forward from the source variable to the variables it affects.

A Calculus of Deviations. A new family of qualitative domains, $P_{B,N}$, was defined. $P_{B,N}$ domains have a logarithmic scale so that a large range of values may be partitioned by relatively few symbols at a coarseness appropriate for the size of the numbers. The parameter B can be changed to alter the size of the ranges that the

symbols represent. The parameter N determines the number of symbols in the qualitative domain. Together, they determine the coverage of the domain (not counting the two extreme symbols that go to infinity.)

An algebra was developed for $P_{B,N}$, including the general functions used by the causality diagrams. The inverse functions were also defined.

The concept of a deviation was defined formally and used to derive deviation formulas for the general functions. The $P_{B,N}$ algebra was then applied to these formulas to produce a calculus of deviations. In addition to the normal and inverse functions of the $P_{B,N}$ qualitative algebra, "assumptive" functions, crucial to propagating deviations with incomplete information, were defined.

Search procedures. Two alternative search algorithms were presented. Both algorithms work from a causality diagram, applying the calculus of deviations in order to propagate the analyst's initial assumptions forward. The SemiAutomated algorithm allows the analyst to control which path the search will take. The analyst's options include following a state forward one step to propagate definite deviations or adding assumptions (provided by the algorithm) to the current state in order to propagate possible deviations.

Given a set of initial assumptions, the Automatedalgorithm performs a "definite-first" search. The algorithm thus provides the most likely scenarios first.

8.2 Future Directions

Causality diagram size. Although chapter 4 showed that the causality diagram is appropriate for state-based languages, the language needs improvement. It is subject to get very large. The causality diagram for a medium-sized RSML specification can contain 5,000 nodes.[1] The issue is one of size versus speed, with the economics of computer resources suggesting that memory can afford growth faster than the CPU can. However, this strategy may need to be modified for very large specifications.

Application to other specification languages. An outstanding research issue is the applicability of the causality language to various specification languages. Chapter 4 showed that state-diagram concepts can be translated to causality diagrams. The causality diagram also appears well-suited to functional definitions.

Timing. A weakness of the language of causality lies in its treatment of timing. The author is confident that the language of causality can be extended to include more complex temporal relations. An obvious extension is to qualify sequential edges with timing information. This strategy would be similar to some Petri-net variants. Although the calculus of deviations would be unaffected by such an extension, the deviation analysis procedures would require significant changes.

Of particular interest is the development of deviations based on timing. The notions of *early* and *lute* are currently defined as sequences of {true-wrong, false-wrong} and {false-wrong, true-wrong}, respectively. These deviations could be refined with additional timing information.

[1] In the author's implementation of the deviation analysis algorithm, the nodes appear to use well under 100 bytes on average, not counting descriptive strings taken from the specification.

Improved search strategy. The first incarnation of deviation analysis employs a "definite-first" search strategy. If a deviation can definitely be propagated, then the algorithm follows that line of inquiry before turning to deviations that require making additional assumptions on the system state. At first blush this appears to be a sound strategy, but more research needs to be conducted into alternative strategies. The analyst is a potential source of additional information to guide the analysis. Perhaps data from usage of the SemiAutomated algorithm could guide future research in this area.

Backward deviation analysis. Deviation analysis was developed as a forward search method. As discussed already, the forward algorithm is very inefficient at finding the effects of particular assumptions on particular critical nodes. A backward deviation analysis, exploiting the backward-definite operation and adding a backward-possible operation, may be a reasonable alternative or supplement to fault tree analysis and other backward methods. This possibility should be researched for its practicality and usefulness.

Calculus of Deviations. The calculus of deviations is currently based on a specific qualitative mathematics, $P_{B,N}$. $P_{B,N}$ can be improved by representing the negative powers $0 < B^{-q} < 1$. This improvement provides a reciprocal balance for multiplication the way that negation does for addition.

Alternatively, a calculus of deviations could be developed for intervals. The author has already begun research into this area. The results can be much more precise than with qualitative mathematics, but the precision comes at a cost. First, the number of intervals could grow quite large, leading to a nonlinear increase in computations (which are already more complex than qualitative calculations.) Second,

analysis over arbitrary intervals is subject to "creep", whereby a node's value is incrementally adjusted over time, leading to much finer-grained analysis. Such an analysis may be too involved to be useful early in software development. The coarse nature of qualitative mathematics allows whole classes of values to be treated at once.

Analysis skills. With deviation analysis, the analyst has been provided with a new tool, but how to use the tool is a significant issue. The analyst needs skills to help focus on the analysis that will help most. This is a difficult problem because it is often the analyst's knowledge that interferes with critical review of the specification. Deviation analysis has concentrated on the mechanizable parts of HAZOP. Perhaps some attention should be paid now to adapting the social component of HAZOP to deviation analysis.

Space and time efficiency. The space and time figures cited in chapter 7 are for unoptimized code. Moreover, although considerable thought went into a time-efficient algorithm, little attention was paid to optimizing for memory requirements. This problem is not trivial, since the algorithm can recognize any state that has been visited along another search path. Every state must thus be kept available for the duration of the search.

In conclusion, deviation analysis appears to be a practical method of requirements analysis. The strength of HAZOP guide words has been captured in part by the calculus of deviations. However, some guide words must be encoded by deviation sequences. The `Automated` algorithm provides meaningful results in a reasonable amount of time. The usefulness of the results will be determined by practitioners.

Bibliography

[1] Aho, Sethi, and Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[2] Richard C. Booten Jr. and Simon Ramo. The development of systems engineering. *IEEE Transactions on Aerospace and Electronic Systems,* AES-20(4), July 1984.

[3] W.C. Bowman, G.H. Archinoff, V.M. Raina, D.R. Tremaine, and N.G. Leveson. An application of fault tree analysis to safety-critical software at ontario hydro. In *Conference on Probabilistic Safety Assessment and Management (PSAM),* April 1991.

[4] R.W. Butler and G.B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering,* 19(1):3–12, January 1993.

[5] Yue-Lung Cheng, Hsiu-Chuan Wei, and John Yuan. On establishment of i/o tables in automation of a fault tree synthesis. *Reliability Engineering and System Safety,* 40:311–318, 1993.

[6] Federal Aviation Administration, 800 Independence Avenue, S.W., Washington, D.C., 20591. *TCAS II Collision Avoidance System (CAS) System Requirements Specification,* change 6.00 edition, March 1993.

[7] Peter Fenelon and John A. McDermid. An integrated tool set for software safety analysis. *Journal Systems Software,* 21:279–290, 1993.

[8] Tom Forester and Perry Morrison. Computer unreliability and social vulnerability. *Futures,* pages 462–474, June 1990.

[9] Sergio Guarro and David Okrent. The logic flowgraph: A new approach to process failure modeling and diagnosis for disturbance analysis applications. *Nuclear Technology,* 67, December 1984.

[10] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming,* 8:231–274, 1987.

[11] Mats Per Erik Heimdahl. *Static Analysis of State-based Requirements.* PhD thesis, University of California, Irvine, 1994.

[12] Danial Hernández. Reasoning with qualitative representations: Exploiting the structure of space. In *QUARDET '93: III IMACS International Workshop on Qualitative Reasoning and Decision Technologies,* June 1993.

[13] Jonathan Jacky. Safety-critical computing: hazards, practices, standards and regulation. In C. Dunlop and R. Kling, editors, *Computerization and controversy,* chapter 5, pages 612–631. Academic Press, 1991.

[14] Stephen A. Lapp and Gary J. Powers. Computer-aided synthesis of fault-trees. *IEEE Transactions on Reliability,* pages 2–13, April 1977.

[15] Nancy G. Leveson. Software safety: Why, what, and how. *ACM Computing Surveys,* 18(2):125–163, June 1986.

[16] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[17] Nancy G. Leveson and Peter R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering,* SE-9(5):569–579, September 1983.

[18] Nancy G. Leveson and Janice L. Stolzy. Safety analysis using petri nets. *IEEE Transactions on Software Engineering,* SE-13(3):386–397, March 1987.

[19] Peter Lewycky. Notes toward an understanding of accident causes. *Hazard Prevention,* March/April 1987.

[20] Bev Littlewood and Lorenzo Strigini. The risks of software. *Scientific American,* 267(5):38–43, November 1992.

[21] Robyn R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE international symposium on requirements eng ineering,* pages 35–46, January 1993.

[22] J.A. McDermid and D.J. Pumfrey. A development of hazard analysis to aid software design, 1994.

[23] Bonnie E. Melhart. *Specification and Analysis of the Requirements for Embedded Software with an External Interaction Model.* PhD thesis, University of California, Irvine, July 1990.

[24] Bonnie E. Melhart and Nancy G. Leveson. A specification model for safety analysis of embedded software. Technical report, University of California, Irvine, September 1990.

[25] A.R.T. Ormsby, J.E. Hunt, and M.H. Lee. Towards an automated fmea assistant. In *Applications of Artificial Intelligence in Engineering VI,* pages 739–752. Elsevier Applied Science, July 1991.

[26] Charles Perrow. *Normal Accidents.* Basic Books, New York, 1984.

[27] Henry Petroski. Galileo and the marble column: A paradigm of human error in design. *Structural Safety,* 11:1–11, 1991.

[28] C.J. Price, J.E. Hunt, M.H. Lee, and A.R.T. Ormsby. A model-based approach to the automation of failure mode effects analysis for design. In *Proceedings of the Institution of Mechanical Engineers Vol. 206,* 1992.

[29] Philip Schaefer. Analytic solution of qualitative differential equations. In *Proceedings Ninth National Conference on Artificial Intelligence,* pages 830–835. AAAI Press, July 1991.

[30] Roland Schinzinger. Technological hazards and the engineer. *IEEE Technology and Society Magazine,* pages 12–16, June 1986.

[31] Jouko Suokas. The role of safety analysis in accident prevention. *Accident Analysis and Prevention,* 20(1):67–85, 1988.

[32] J.R. Taylor. Sequential effects in failure mode analysis. In *Reliability and fault Tree Analysis,* pages 881–894, 1975.

[33] J.R. Taylor. An algorithm for fault-tree construction. *IEEE Transactions on Reliability,* R-31(2):137–146, June 1982.