

# Safety Analysis Tools for Requirements Specifications\*

Vivek Ratan  
Kurt Partridge  
Jon Reese  
Nancy Leveson

## Abstract

This paper describes safety analysis 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, tools to check for consistency and completeness of specifications, and additional safety analysis techniques. An example requirements specification for an Automated Highway System (AHS) is used for describing the functionality of the tools.

**Keywords:** Software engineering, software safety, hazard analysis, fault tree.

## 1 Introduction

The goal of the University of Washington's Safety-Critical Software project is to develop a theoretical foundation for software safety, to build a methodology upon that foundation, and to provide tools and techniques for automating support of safety analysis. The techniques and tools share a common model-based interface that facilitates communication between members of a project development team—managers, application experts, requirements writers, designers, developers, safety analysts, testers, and potential operators. We

---

\*This work was partly funded by NASA/Langley, NSF, and 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 paper reflect the views of the authors who are responsible for the facts and accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California.

are refining the techniques through iterative feedback from developers of large, safety-critical systems in order to improve the tools' usability and to ensure that the techniques scale up to realistic systems.

Our general approach, called *safeware engineering* [8], involves identifying software hazards and applying software hazard analysis and hazard control procedures throughout the software development process. We have been defining and evaluating specific techniques for implementing the safeware engineering methodology and building prototype tools.

Because safety engineers have concluded that inadequate design foresight and requirements specification errors are the greatest cause of software safety problems [8] and because few tools exist to support this stage of development, we have focused on building an environment for computer-based systems to aid in modeling and analysis during overall system design and software requirements specification. The environment (Figure 1) acts as a workbench for system engineers, software engineers, and human factors experts and can enhance communication among them by using common models and analysis tools that execute on the models.

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 practical 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.

The rest of the paper is organized as follows. Section 2 describes an automated highway system that is used as an example in this paper. Section 3 presents the basic features of RSML and our model of the automated highway system. Finally, Section 4 describes the safety analysis techniques, including forward and backward simulation, generation of fault trees, consistency and completeness analysis, and deviation analysis.

## 2 The Automated Highway System Example

In this paper, we use a requirements specification of an automated highway system (AHS) based on a model developed at the California PATH program at the University of California, Berkeley [5, 6] as an example. This specification, along with several others, is being used as a testbed to explore automated analysis techniques, and as such does not represent any real

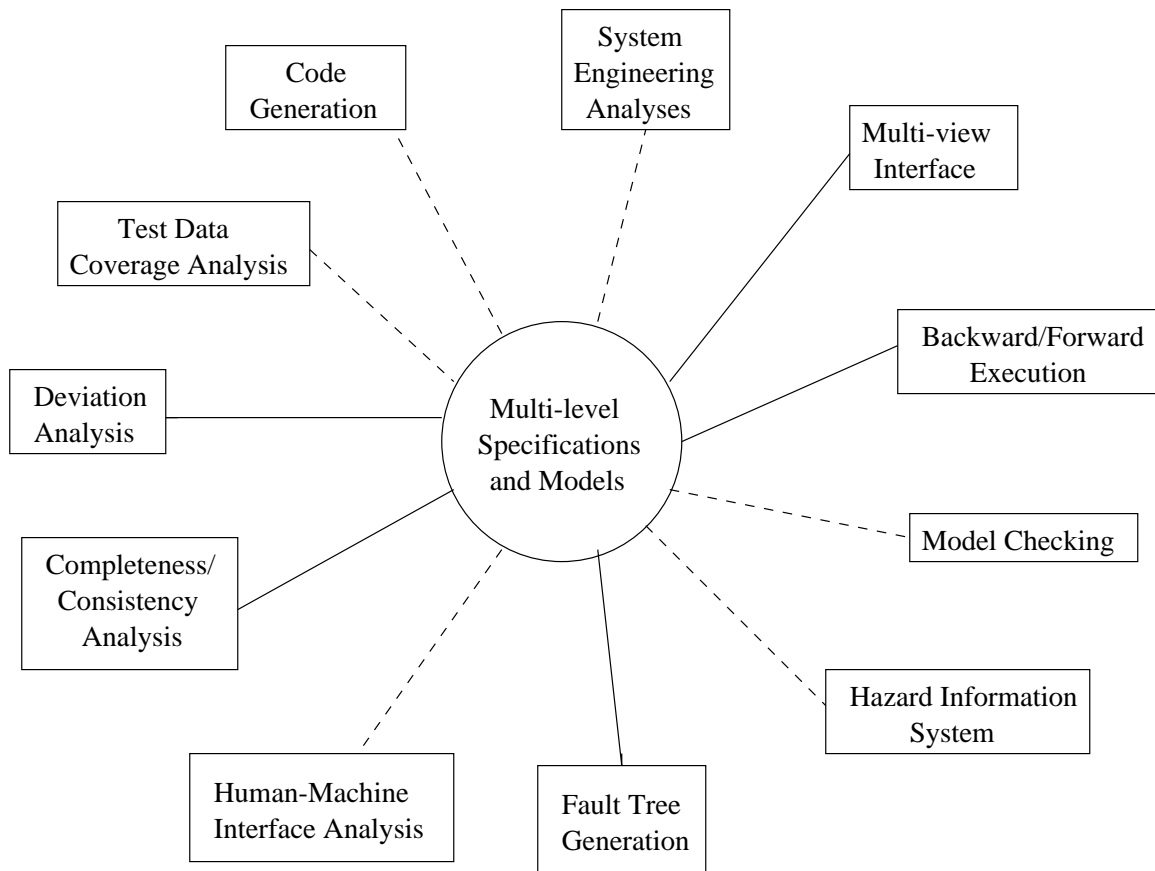


Figure 1: Using a common specification (based on an underlying state-machine model) in the RSML language, the analyst can perform a variety of analyses. The dotted lines represent tools still under development that are not described in this paper.

or complete automated highway design. The example used concentrates on modeling the motion of vehicles on the automated highways. Different and more complete specifications can be written and analyzed as appropriate.

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*. These maneuvers are described shortly.
- On the highway, all vehicles move at the same speed except when they take part in one of the above-mentioned three maneuvers.
- Roadside control structures are present along the highway. They measure traffic conditions and communicate with vehicles, asking them to initiate maneuvers when required.
- 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 be separated from the platoon in front of it by some minimum distance. 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 achieve optimal capacity and reduce travel times of vehicles on the highway.
- Vehicles have the ability to communicate with each other. Vehicles are provided information on their speed and position, as well as the position of other vehicles around them, through their sensors and through communication with the roadside control structures.

There are three kinds of elementary maneuvers that a vehicle can perform on an 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. Again, we note that specific AHS designs may define these maneuvers differently, but the definitions used here are adequate for demonstrating the modeling language and analysis tools.

### 3 RSML and 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. Note that we use RSML as a blackbox requirements specification language for process control systems, that is, the required behavior of the component is defined completely in terms of its externally visible behavior. This blackbox behavior is defined in terms of states and transitions of the controlled process. 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 phase as the understanding of the environment and the controller behavior changes.

RSML models of independent system 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.

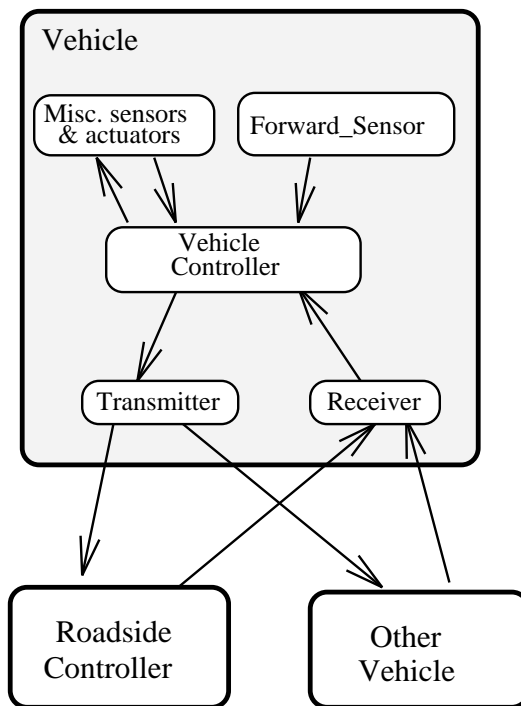


Figure 2: *Communicating components in the AHS model. Arrows represent communication between components. The model in this paper describes the required behavior of the vehicle*

		<i>OR</i>	
<i>A N D</i>	Position <b>In State</b> Single	·	T
	Distance <b>In State</b> IP	T	T
	ThisLaneFront	T	·
	OwnSpeed = System_Speed	F	T

Figure 3: An *AND/OR* table.

Within a vehicle in the AHS, the sensor and the receiver components provide inputs to the vehicle controller, while the vehicle controller provides inputs to the transmitter, which in turn communicates with the receiver on other vehicles. This communication structure is shown in Figure 2. Within a component, internal events are broadcast and available everywhere. In this paper, we consider only the model of the vehicle controller.

Each transition in the RSML state-machine model 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 a rich language for 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 3. 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 in observing the use of RSML by application experts that *AND/OR* tables provide a more natural and reviewable notation when compared to other notations like predicate calculus. 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 *n*th last time that a variable was assigned or event was triggered. Thus, the condition represented by the *AND/OR* table in Figure 3 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`. The type of each variable is denoted by subscripts (which have been omitted in the figure).

In the RSML model of an AHS shown in Figure 4, the controller component is modeled as four parallel state-machines. The complete model is about 50 pages long, and therefore we can only briefly describe it in this paper.

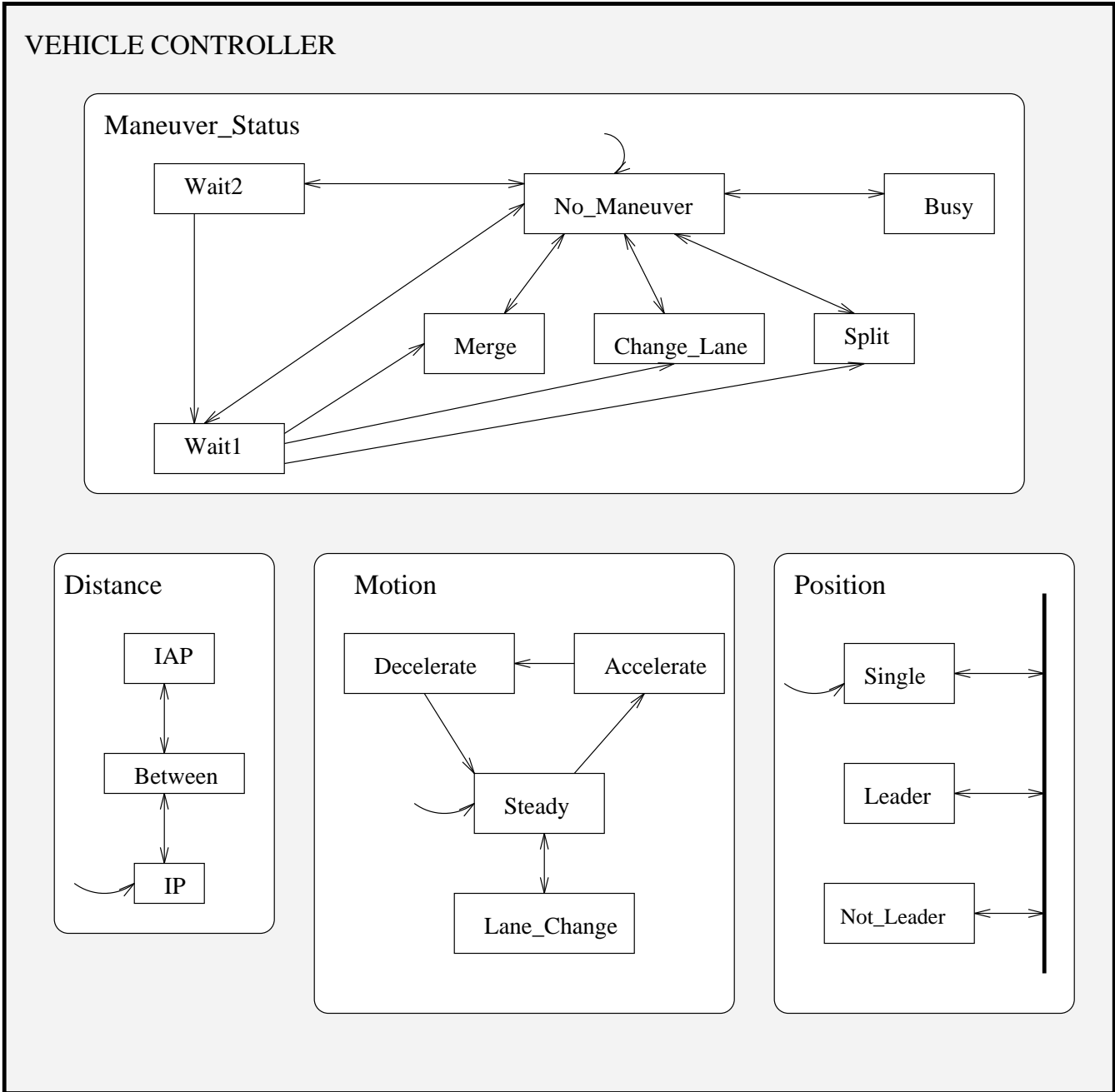


Figure 4: *The state-machines comprising the vehicle controller component. Arrows between states denote the presence of at least one transition between them.*

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,
- `Wait1`: 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.

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.

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). The complete specification can be found in [14].

## 4 Analysis Tools

The goal of our project is to explore the limits of automated analysis to provide information useful in safety-critical project development. We have previously developed algorithms to allow certain 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. These tools and analysis techniques include forward simulation, backward hazard analysis using fault trees to display the results, completeness checking, and software deviation analysis.

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. 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 in general to the way the system is supposed to work. But the large number of potential paths makes such forward simulation fairly ineffective as a hazard analysis tool.

## 4.2 Backward Analysis

Whereas forward analyses start with events and determine whether they can lead to hazardous states, backward analysis starts with a hazardous state and determines what conditions or events could lead to it. When the state space is large and the number of hazards is limited (as is almost always the case), backward analysis may be more effective and feasible than forward analyses.

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. Although theoretically possible to generate the entire reachability graph for these types of models, this is often impractical. Instead, we devised an algorithm that requires looking at what will usually be a small part of the graph to obtain the information necessary to eliminate or control hazards in the design.

Briefly, the procedure starts with a set of hazardous conditions, which will be only partial states (partial configurations) in the reachability graph. Some conditions in the state are unimportant as far as safety is concerned, and, 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.

For each member of this set of hazardous conditions, 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 nonhazardous 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 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.

We have translated this analysis to RSML models and developed a tool to assist with the analysis by performing a backward simulation of the model. The results of this analysis are displayed using a fault tree format.

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 (inadvertent) 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. 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 [16]. 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.

Our procedures are similar but use a system or software model. Automatic fault tree generation from an RSML specification is based on a backward simulation of the system. Backward simulation involves finding previous configurations, that is, those from which there are transitions to the current configuration that take one microstep. For each such "one-step-back" configuration found, fault tree templates (representing mini-fault-trees) are created. These templates represent 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.

Generating the entire fault tree in this manner is again impractical for most complex systems. An analyst may need to apply application expertise to determine which paths are most important to explore. In addition, the algorithms for reducing search that we developed for Petri nets can be applied to RSML models. A particular hazardous configuration (or partial configuration) can have zero or more previous configurations such that a set of parallel transitions can cause the system to move from each of these configurations to the

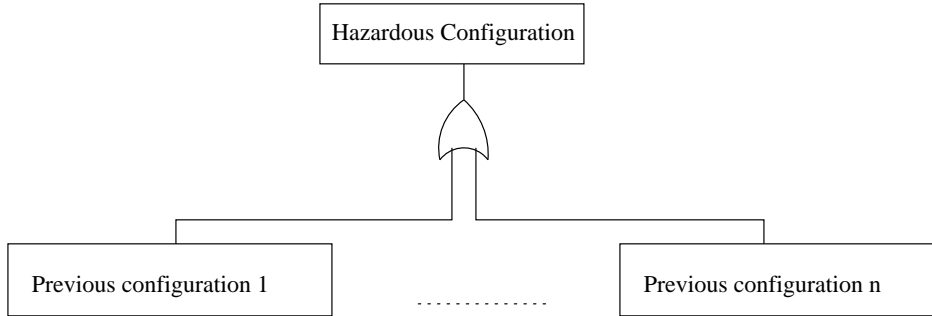


Figure 5: *Basic template for each level of fault tree.*

hazardous configuration in a microstep. For every one-microstep-back configuration constructed, the algorithm considers other configurations that can be reached in one forward microstep. The information obtained can be used to eliminate paths to hazardous states.

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. Finally, we show how fault tree analysis can be used to modify the 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.

Figure 5 shows the basic template for displaying previous configurations. Figure 6 shows 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. 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. Node B contains event sequencing information: The triggering events in this set may themselves need to be triggered prior to other events (a particular ordering of events may be necessary) in order for the system to move to the hazardous configuration. Finally, the guarding condition(s) on each of the orthogonal

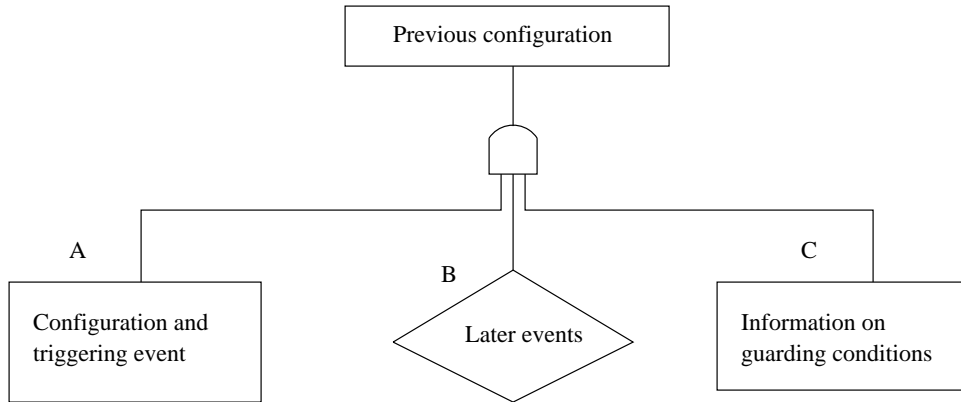


Figure 6: *Expansion of a previous configuration node.*

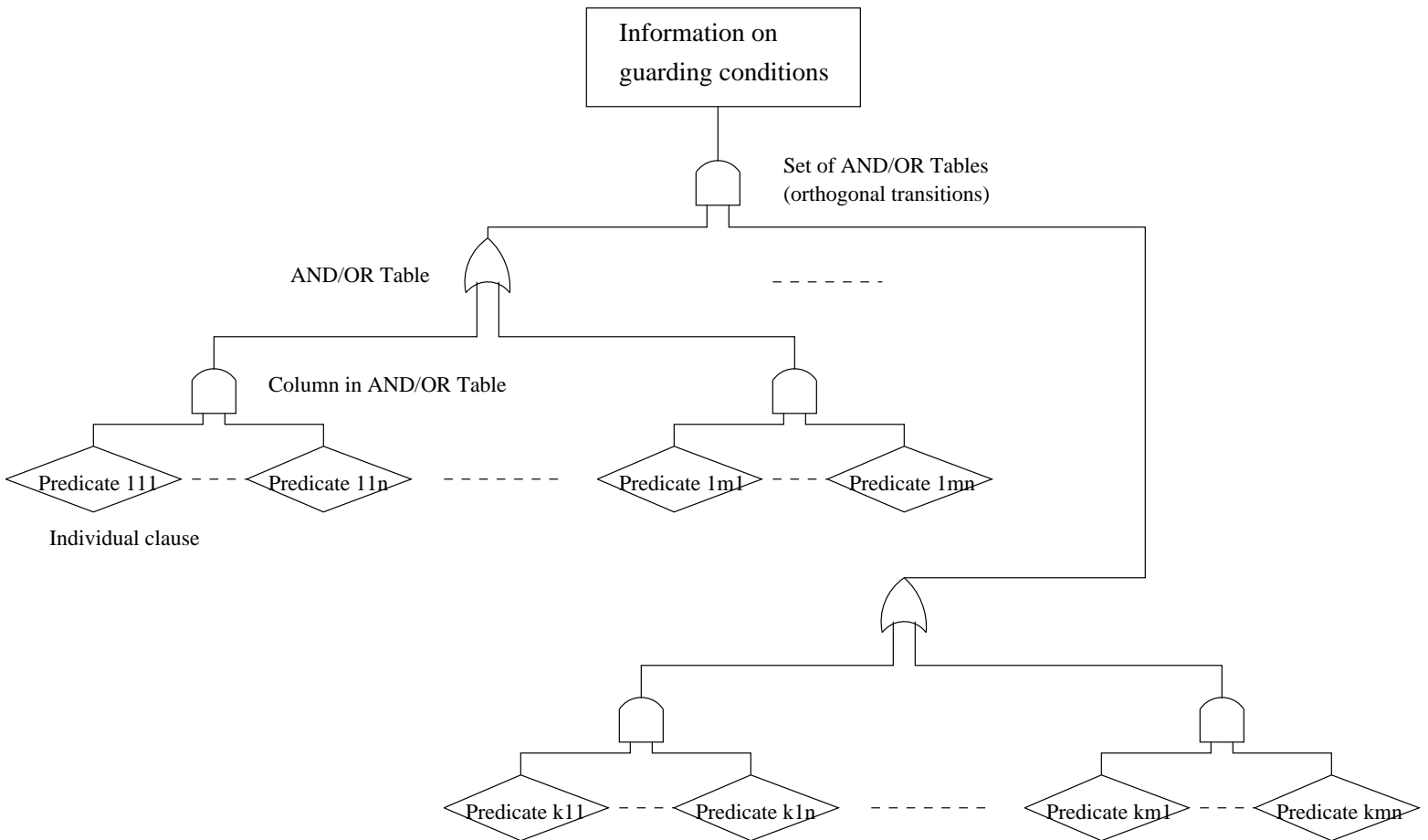


Figure 7: *Expansion of guarding conditions information node.*

transitions, if any, need(s) to be satisfied in order for the transitions to be valid (node C). Figure 7 shows a template for the expansion of node C; it represents the set of guarding conditions that are to be satisfied, in terms of each individual guarding condition.

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

### 4.2.2 Fault Tree Example

A safety constraint of the AHS is that platoons must be at least an inter-platoon distance apart from each other. A hazard arises when this condition is violated as represented by the partial configuration  $C$ , ( $D:Between$ ,  $P:Leader$ ). The fault tree, with  $C$  as root, is shown in figure 8. (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 provides a simple and fast way to display a graph. 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. 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 4.2.3 describes another such situation and suggests a way to modify the design to incorporate a fault-handling mechanism.

### 4.2.3 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  $P$  defined as ( $D:IAP$ ,  $M:Accelerate$ ,  $P:Leader$ ). Figure 9 displays the fault tree with  $P$  as its root (node A). Only one one-step-back configuration ( $P_1$ ), defined as ( $D:Between$ ,  $M:Accelerate$ ,  $P:Leader$ ) and represented by node B, was found by the fault tree generator.  $P_1$  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  $P_1$  to  $P$  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

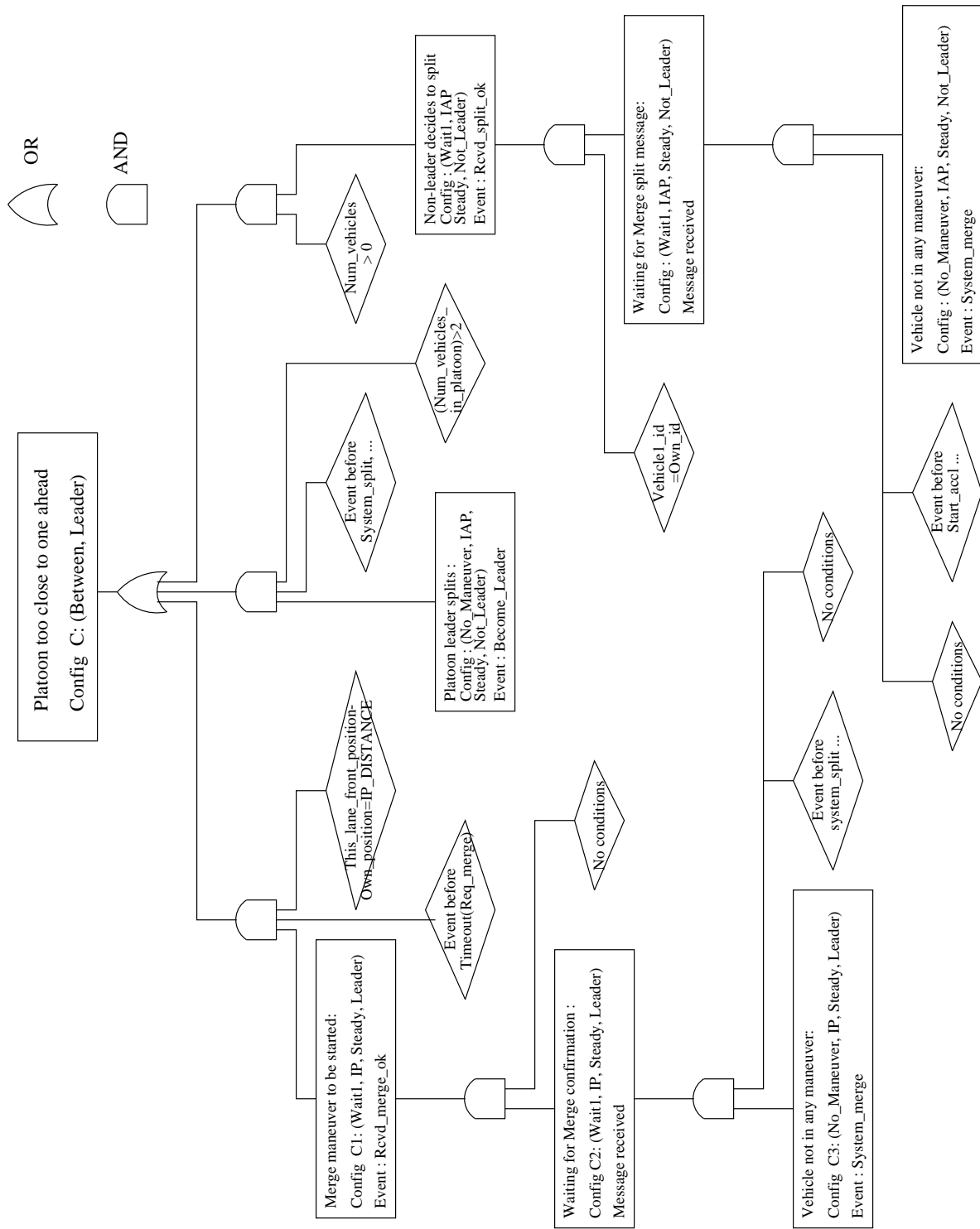


Figure 8: An example of an AHS fault tree generated from an RSML specification.

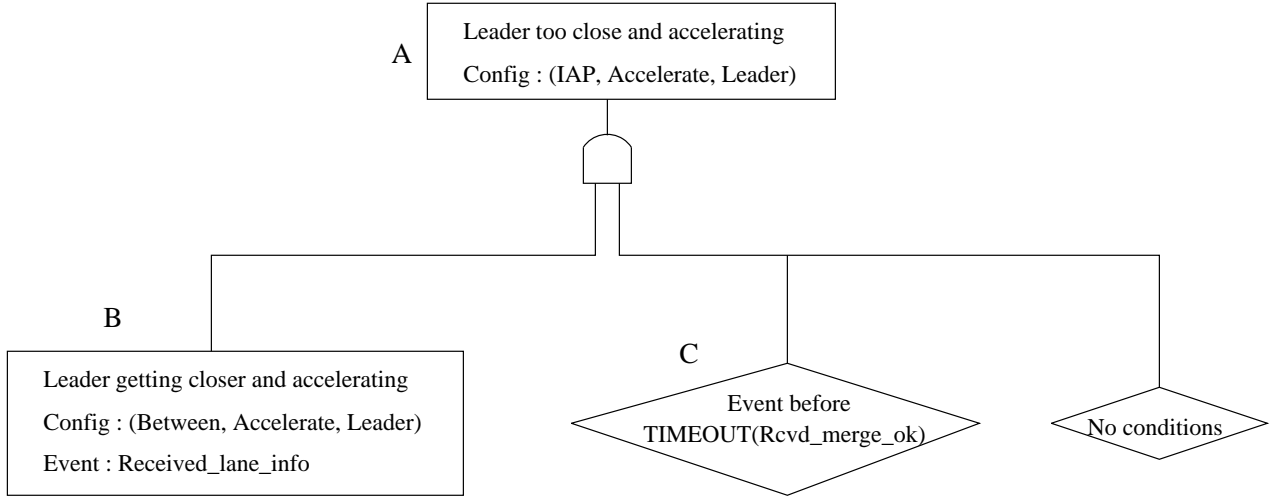


Figure 9: *Fault tree example.*

**Distance** changes state through the transition  $[\text{Between} \rightarrow \text{IAP}]$ , reflecting the sensor input. Node P shows that the system can move from  $P_1$  to  $P$  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  $[\text{Accelerate} \rightarrow \text{Decelerate}]$  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  $[\text{Between} \rightarrow \text{IAP}]$  that causes the system to move from  $P_1$  to  $P$ . However, node P 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  $P$ , 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  $[\text{Accelerate} \rightarrow \text{Decelerate}]$  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 to eliminate hazardous path sequences using interlocks. In the AHS model, we would like the transition  $[\text{Accelerate} \rightarrow \text{Decelerate}]$  to be taken before the transition  $[\text{Between} \rightarrow \text{IAP}]$ . 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.

The fault tree analysis thus enables the designer to detect situations where failures can lead to accidents. Adequate failure-handling mechanisms can then be added to enable the system to deal with the failures.

### 4.3 Robustness Analysis

We have also developed tools to check some aspects of robustness of RSML specifications and to detect nondeterminism in the specification [4].

The output of the tool is a list of conditions on the transitions out of a state that allow more than one transition to be satisfied simultaneously. Performing this analysis on our AHS model, two nondeterministic situations were detected, both arising during the beginning of *Change-lane*.

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`  $\longrightarrow$  `Change_Lane`] is governed by the following two guarding conditions :

- 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 `No_Maneuver` to `Change_Lane` can be taken, along with either of the transitions from `No_Maneuver` 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.

### 4.4 Deviation Analysis

Deviation analysis is a new forward analysis technique developed by Reese [15] that takes its inspiration from HAZOP (HAZards and OPerability analysis), a very successful analysis procedure in the chemical process industry. Both techniques are based on the underlying system theory that considers accidents to be the result of deviations in system variables.

In order to reason about the effects of deviations, the analysis borrows from a relatively recent area of research commonly called “qualitative mathematics,” which operates on categories of numbers rather than the numbers themselves. Reese has developed a “calculus of deviations” and a causality diagram to serve as foundations for a forward search algorithm.

Nodes in the causality diagram correspond to system variables in the specification. Additionally, each node is associated with a function defined by the calculus of deviations, so that system variable deviations may be propagated qualitatively by applying the functions.

A translator has been written to build causality diagrams from RSML specifications and to perform the deviation analysis. To use the tool, the analyst provides some initial assumptions about the system state, including at least one deviation, and also a list of safety-critical system variables. The algorithm returns a list of scenarios that begin with the analyst's assumptions and lead to a deviation in one of the safety-critical variables.

As an example, one of the inputs to the vehicle controller is the position of the vehicle in the platoon. We will assume that the analyst is concerned with the value of this input being too high, and we will also assume that all of the controller's outputs are considered to be safety-critical, although the analyst could instead choose to check a subset of these outputs or even some system variables downstream of the computer.

Given these assumptions, the deviation analysis algorithm applied to the AHS model produced five scenarios. In one of the scenarios, the algorithm had to make four assumptions about the actual values of system variables. These assumptions, combined with the initial deviation, lead to a request for the vehicle to change lanes when it should not. For two other scenarios the deviation analysis algorithm generated assumptions that include new deviations, resulting in a situation where lane-change requests are supposed to occur but do not. Two other scenarios resulted in two different platoon split commands not being communicated properly. Note that the deviation analysis tools can examine single or multiple independent deviations.

The strength in deviation analysis lies not so much in finding out whether an input deviation will lead to an output deviation (it almost surely will) but in presenting various ways that the deviation is propagated and in determining which deviations will lead to hazardous outputs.

## 5 Conclusions

This paper has described tools for the safety analyses of RSML specifications. These tools include a forward simulator, a fault tree generator based on backward simulation, a robustness and nondeterminism checker, and a software deviation analysis tool.

Although the tools work for the example model, further experimentation is required to determine whether the procedure is practical and useful for other examples and whether it can provide information that is important to the system designers. Of particular concern are problems associated with state-space explosion. Additional manual and automated pruning methods may be needed to reduce the number of previous states considered.

## 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. *IEEE Trans. on Software Engineering*, May, 1996.
- [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 Transactions 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 Transactions 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] Vivek Ratan, Kurt Partridge, Nancy Leveson. Safety Analysis Tools for AHS Models. Submitted as a PATH Technical Report, January 1996.
- [15] Jon Damon Reese. *Software Deviation Analysis*. Ph.D. Dissertation, University of California, Irvine, 1996.
- [16] 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.