Software Deviation Analysis: A "Safeware" Technique*

Jon Damon Reese and Nancy G. Leveson

Dept. of C.S.E.
University of Washington
Box 352350
Seattle, WA 98195, U.S.A.

Safeware Engineering Corp.
7200 Lower Ridge, Unit B
Everett, WA 98203, U.S.A.

{jdreese,leveson}@cs.washington.edu
{jdreese,leveson}@safeware-eng.com

Abstract

Standard safety analysis techniques are often ineffective when computers and digital devices are integrated into plant control. The *Safeware* methodology and its set of supporting safety analysis techniques (and prototype tools) includes modeling and hazard analysis of complex systems where the components may be a mixture of humans, hardware, and software. This paper describes one of the Safeware hazard analysis techniques, Software Deviation Analysis, that incorporates the beneficial features of HAZOP (such as guidewords, deviations, exploratory analysis, and a systems engineering approach) into an automated procedure that is capable of handling the complexity and logical nature of computer software.

*This work was partly funded by NASA/Langley Grant NAG-1-1495, NSF Grant CCR-9396181, and the California PATH Program of the University of California, in cooperation with the California Department of Transportation and the U.S. Department of Transportation.

1 Introduction

The introduction of computer control in plants has created new and unsolved problems in ensuring safety. For the past 16 years, Leveson and students have been studying ways to extend and adapt to software the methods used to control risk in the larger system within which the software is embedded. Although engineers have developed various types of hazard analysis techniques for electromechanical systems, these techniques do not apply when computers are introduced to control dangerous and complex systems. Our goal is to take the basic procedures of system hazard analysis and to translate them into techniques and tools that can be applied to systems containing software and to the software development and validation process.

This work has resulted in an approach, called Safeware, to enhance safety in systems composed of electromechanical, computer, and human components. The basic methodology involves applying software hazard analysis and hazard control procedures throughout software development, based on the identified system hazards [2]. These efforts are closely tied to the system level hazard analysis and control. Early and continuing analysis procedures guide and direct the software as it is developed instead of simply attempting to verify safety after the software is completed. The methodology includes a variety of analysis techniques and tools (see Figure 1). This paper describes one technique, Software Deviation Analysis, that extends basic HAZOP-like analysis to systems containing computer components.

2 Software Deviation Analysis

HAZard and OPerability analysis (HAZOP), a review procedure developed for the British chemical industry in the 1950's, is used widely but it has several limitations when applied to newer, high-technology systems. First, it is time- and labor-intensive [2], in large part due to its reliance on group discussions and manual analysis procedures. Second, HAZOP analyzes causes and effects 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.

A third limitation arises from the fact that HAZOP is a flow-based analysis. Deviations from within components or processes are not inspected directly; instead, a deviation within a component (as well as a human error or other environmental disturbance) is assumed to be manifested as a disturbed flow [4]. A purely flow-oriented approach may cause the analyst to neglect process-related malfunctions and hazards in favor of pipe-related causes and effects.

Because HAZOP concentrates on physical properties of the system [4], it is not directly applicable to analyzing computer input and output. Several manual techniques have been suggested to extend HAZOP to incorporate inspection of computer hardware and software. In each of these, the procedure is essentially identical to a standard manual HAZOP except that the guide-words are changed and the model of the system

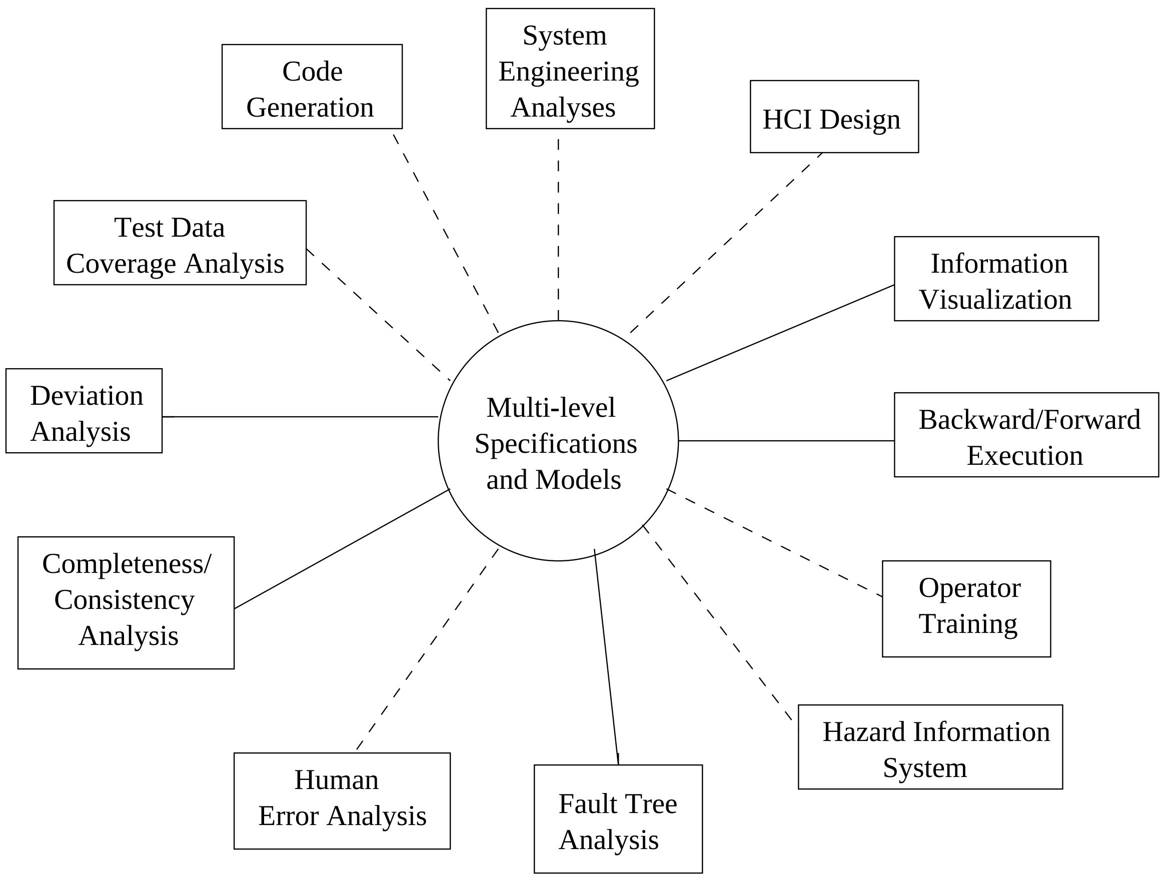

System Engineering Analyses
Code Generation
HCI Design
Test Data Coverage Analysis
Information Visualization
Deviation Analysis
Multi-level Specifications and Models
Backward/Forward Execution
Completeness/ Consistency Analysis
Operator Training
Hazard Information System
Human Error Analysis
Fault Tree Analysis

Figure 1: Components and interactions in a CAD environment for safety-critical systems.

may differ from the original pipe-and-process diagram.

These methods suffer from two weaknesses with respect to analyzing software. First, being manual techniques they depend on human understanding of the proposed software, which can be quite limited. Whereas the components of a pipe-and-process diagram usually conform to straightforward and well-understood laws, each instance of a software controller can have a complex and novel behavior: The behavior of pumps and valves is not nearly as complex as software can potentially be.

Second, the manual techniques adhere to the HAZOP principle of identifying deviations in the connections, i.e., the computer inputs and outputs only. Accordingly, they do not provide guidance for following deviations into the control logic. The task of determining how a deviation in a software input is manifested at its outputs is left wholly up to the analyst.

Software Deviation Analysis overcomes some of these deficiencies. Like HAZOP, Software Deviation Analysis is based on the underlying model that accidents are caused by deviations in system parameters. Using a blackbox software or system requirements

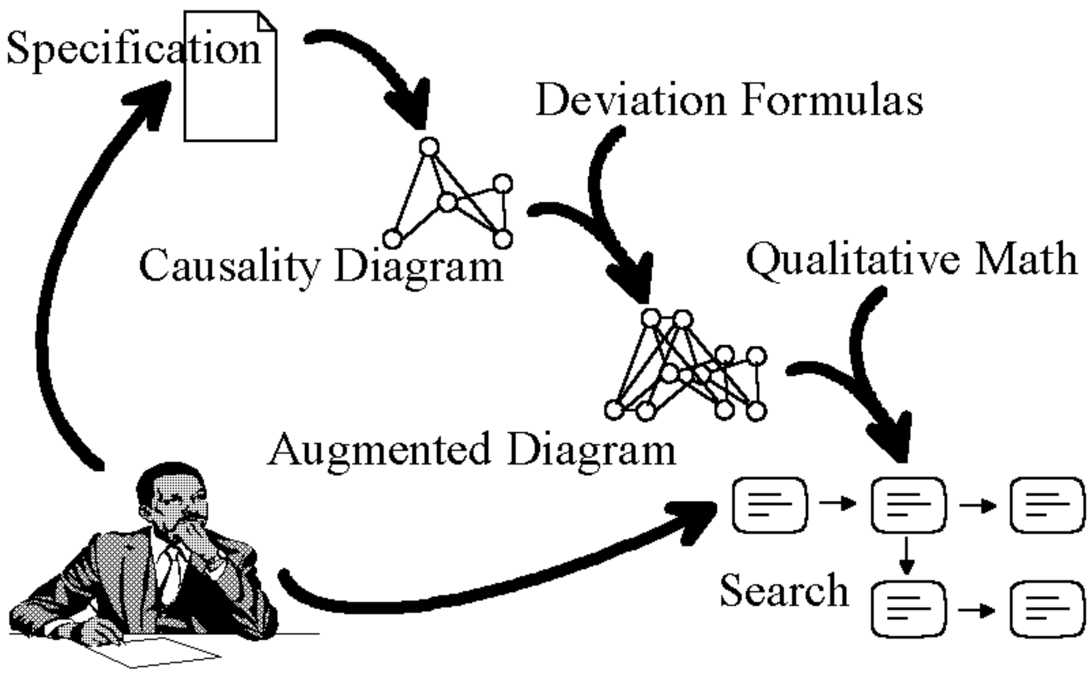


Figure 2: Overview of SDA procedure.

specification, the analyst provides assumptions about particular deviations in software inputs and hazardous states or outputs, and the procedure automatically generates scenarios in which the analyst's assumptions lead to the specified deviations in the outputs.

Figure 2 shows an overview of the SDA procedure. The analyst provides a formal software requirements specification, which the procedure automatically converts into a more basic representation, called a *causality diagram.* The causality diagram is an internal data structure that encodes causal information between system variables, based on the specification and the semantics of the language in which it is written. The simplicity of causality diagrams makes the search algorithm more straightforward and easier to adapt to a new specification or programming language. Causality diagrams may also be helpful to the analyst in understanding how system variables are interrelated.

The automated procedure uses *deviation formulas*, which define how deviations are related. This information is incorporated directly into the causality diagram to create an *augmented* causality diagram. SDA then uses qualitative mathematics on the augmented causality diagram to evaluate deviations. Qualitative mathematics partitions infinite domains into a small set of intervals and provides mathematical operations on these intervals. The use of fixed intervals simplifies the analysis compared to iterations over the entire state space. It also lends itself naturally to the qualitative nature of deviations, such as "slightly too high."

The augmented causality diagram, input deviations, and a list of safety-critical software outputs is passed to the search program, which constructs a tree of states. The state formed by the input deviations is the root of the search tree. Leaves are either deadend searches (in which the state does not contain any deviations) or states containing safety-critical deviations.

The output of the SDA program is a list of *scenarios*. A scenario is a set of deviations in the software inputs plus constraints on the execution states of the software that are sufficient to lead to a deviation in a safety-critical software output. The deviation analysis procedure can optionally add further deviations as it constrains the software state, allowing for the analysis of multiple independent failures (leading to the independent deviations.)

The next sections describe the components of deviation analysis—causality diagrams, deviation formulas, qualitative mathematics, and the analysis procedure. Section 7 provides a simple example.

3 Causality Diagrams

Figure 3 shows an example of a simple feedback system and the corresponding causality diagram. A requirements specification for the software controller contains a black-box model of the relationship (function) between inputs (the measured system variables) and outputs (commands to change the controlled system variables).

The system shown in the figure 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 computed as 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. *Pressure* is a quotient function, with the numerator edge originating from *Flow* and the denominator edge originating from *Aperture*. *Aperture* is an interesting node in that its current value depends on its previous value, i.e., it has state. 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 behavior of the *Flow* variable is left unspecified. The procedure treats unspecified variables as random inputs to the system and constrains them as needed to propagate deviations.

The remaining nodes comprise the controller (which for process-control software would be a computer, usually performing a much more complex function.) The pres-

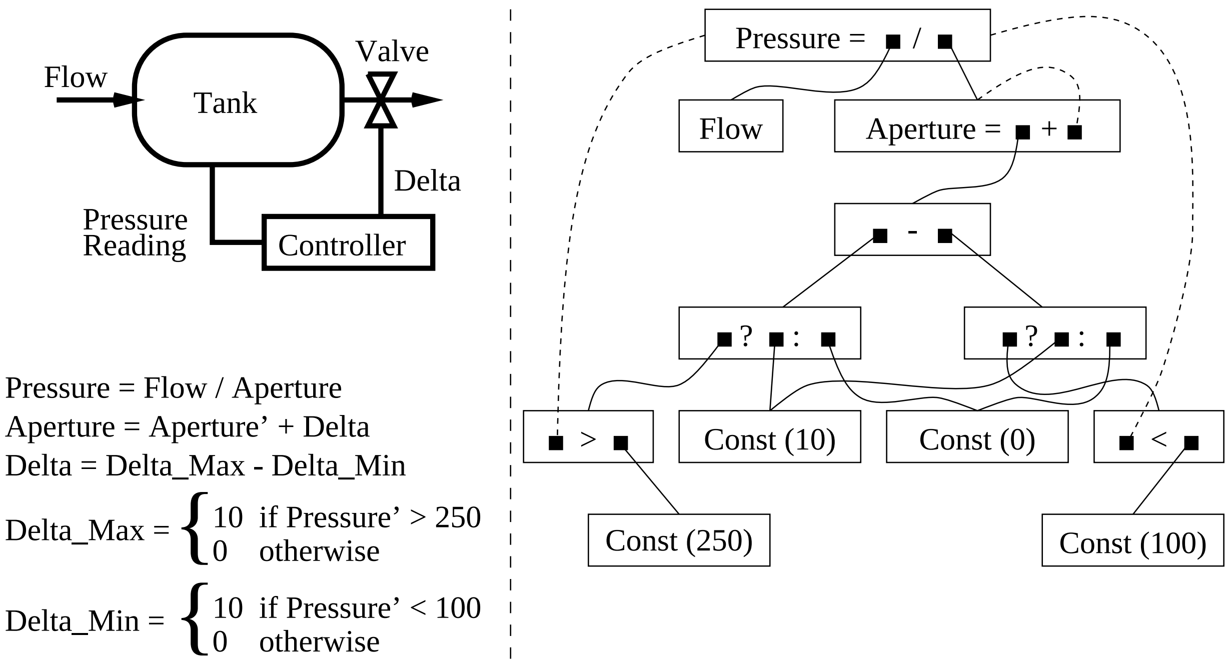

Flow
Tank
Valve
Delta
Pressure Reading
Controller
Pressure = Flow / Aperture
Aperture = Aperture' + Delta
Delta = Delta_Max - Delta_Min
Delta_Max = 10 if Pressure' > 250; 0 otherwise
Delta_Min = 10 if Pressure' < 100; 0 otherwise
Pressure = ■ / ■
Flow
Aperture = ■ + ■
■ - ■
■ ? ■ : ■
■ ? ■ : ■
■ > ■
Const (10)
Const (0)
■ < ■
Const (250)
Const (100)

Figure 3: Causality diagram example

sure reading is compared to minimum and maximum values (the "<" and ">" nodes, respectively). Note that each node is a function: For example, 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:

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

The selection function is an important node in constructing the causal relationships of process-control software, since it maps from a boolean value (e.g., some control decision) to numeric values (e.g., output to an actuator.)

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

$$\left\{ \begin{array}{ll} 10 & \text{if } \mathit{Pressure'} > 250 \\ 0 & \text{otherwise} \end{array} \right\} - \left\{ \begin{array}{ll} 10 & \text{if } \mathit{Pressure'} < 100 \\ 0 & \text{otherwise} \end{array} \right\}$$

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

The causality diagram can get complex, but it is generated automatically from a state machine specification. Each node in the specification is linked to its corresponding node in the causality diagram, so that the results of the analysis can be translated from the causality diagram back into the language of the specification, avoiding the need for the analyst to comprehend or even see the causality diagram.

4 Deviation Formulas and Augmented Diagrams

The automated procedure uses *deviation formulas*, which define how deviations are related. This information is incorporated directly into the causality diagram to create an *augmented* causality diagram.

The concept of a deviation needs to be defined both for logical and numeric variables. Booleans are straightforward since they can only take two values; consequently, an actual value is either a deviation from the correct value or it is not. The "exclusive OR" operator satisfies this definition. $Actual \oplus Correct$ is *true* (a deviation) when *Actual* is different from *Correct* and *false* when they are the same.

A numeric deviation is defined as the difference between the actual and correct values, i.e., the amount added to or subtracted from the correct value to obtain the actual value:

$$X_d = X_a - X_c,$$

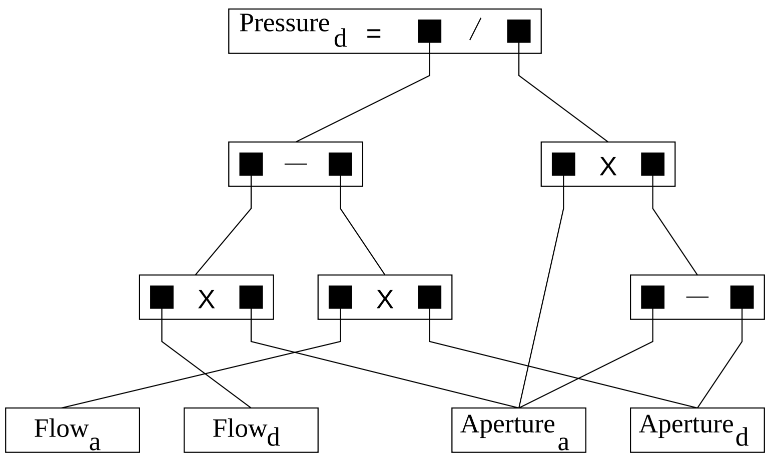
where X is the variable, and the subscripts indicate deviation, actual, and correct values, respectively.[1] For example, if a pressure reading should be 10 p.s.i. but is actually 7 p.s.i., then the deviation is -3 p.s.i.

To relate these values back to the causality diagram, one could assign some correct values and use the relationships expressed in the causality diagram to derive other correct values. Actual values can be derived from other actual values in the same way. For example, $\texttt{Pressure}_a = \texttt{Flow}_a/\texttt{Aperture}_a$.

Deviation values cannot be calculated using the normal relationships between system variables: $\texttt{Pressure}_d$ is not always equal to $\texttt{Flow}_d/\texttt{Aperture}_d$. (In fact, if the actual value of `Aperture` is correct, then $\texttt{Aperture}_d = 0$ and this ratio is undefined.) The causality diagram must be augmented with deviation formulas so that the relationships between deviations are explicitly and properly represented. A deviation formula is the way by which the deviations of a function may be determined from the deviations and actual values of its inputs.

A complete set of deviation formulas and their derivations can be found in [3]. The augmented causality diagram of the tank example is rather larger than the original diagram, and the reader will not be burdened with a complete example, but the fragment representing $\texttt{Pressure}_d$ is shown in Figure 4.

[1]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 definition, a value of $X_d = -0.5$ would mean that X_a has 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 $X_a = 0$.



$\text{Pressure}_d = \blacksquare / \blacksquare$
$\blacksquare - \blacksquare$
$\blacksquare \times \blacksquare$
$\blacksquare \times \blacksquare$
$\blacksquare \times \blacksquare$
$\blacksquare - \blacksquare$
Flow_a
Flow_d
Aperture_a
Aperture_d

Figure 4: A fragment of the augmented causality diagram for the example, showing the causal relationship between deviation in pressure and the deviations and actual values of flow and valve aperture.

5 Qualitative Mathematics

SDA uses qualitative mathematics on the augmented causality diagram to evaluate deviations. Qualitative mathematics partitions infinite domains into a small set of intervals and provides mathematical operations on these intervals. The use of fixed intervals simplifies the analysis compared to iterations over the entire state space. It also lends itself naturally to the qualitative nature of deviations, such as “slightly too high.” The analyst may choose to refer to qualitative set elements, to the intervals they represent, or to symbolic labels such as “a little low,” “too high,” or “very high.”

In a manual HAZOP, analysts need to determine the result of a particular deviation. To do this, they investigate what will definitely occur given the input deviation as well as what else could occur under specific conditions—i.e., they make assumptions about the system state. They must also back-track occasionally to determine whether two separate assumptions are consistent, i.e., whether the scenario they are proposing is realistic. These three types of manual search are also included in the SDA procedure as three functions for each operator appearing on causality diagram nodes: a *forward definite* function, a *backward definite* function, and a *forward assumptive* function.

The forward definite function is simply the operator itself, but applied to sets of intervals. Allowing operations over sets of intervals rather than single intervals greatly reduces the size of the search tree, since a separate branch does not have to be created for each combination of values.

The forward assumptive function is used only on deviation nodes (the nodes created for the augmented diagram). If one of the inputs to a node is a deviation but the value of the node itself is unknown, then the forward assumptive function attempts to find values for the other inputs that will cause the deviation to propagate to the node. For example, suppose that a deviation node is the product of two other nodes, one of which is too high and the other is unknown. In order to propagate the high value, the unknown input is assumed to be negative or positive, but not zero, because a zero will cause the output to be zero, masking the deviation. With the new constraint the output is now either too high or too low.

The backward definite function is essentially the inverse relation of a node's operator. See Reese [3] for definitions of these relations.

The augmented causality diagram, input deviations, and list of safety-critical variables is passed to the search algorithm, which constructs a tree of states. The state formed by the input deviations is the root of the search tree. Leaves are either dead-end searches (in which a state does not contain any deviations) or states containing safety-critical deviations. The output of the SDA procedure is a list of the paths from the root state to all leaves with safety-critical output deviations.

6 Analysis Procedure

The SDA procedure is a forward search procedure—it starts with a deviation in the software's input environment and attempts to find ways in which the deviation can lead to hazardous software outputs. As discussed in the overview, the analysis starts from a system specification, which is converted automatically into an augmented causality diagram. The analyst provides two other pieces of information corresponding to the starting and ending points of the search: (1) an initial system state, including at least one deviation, and (2) the outputs that are safety-critical. The procedure searches forward from the initial state, attempting to find states in which a safety-critical output deviation occurs.

The search procedure is quite complex, and we can only describe it briefly here. The forward and backward definite functions described earlier are used to construct a chain of states representing what will definitely result from the initial state. This chain of states is termed a *scenario* because it describes a sequence of events that the system will follow given the initial state. The chain begins with the initial state and terminates with a state that either contains a safety-critical deviation or no deviation at all. A final state that contains a safety-critical deviation indicates that the analyst's input deviations will always result in a hazardous deviation.

Whether or not the chain leads to a hazardous deviation, the procedure can continue the search by constraining the software state using the forward assumptive function described earlier. Constraints can be added not only to the initial state but to every state in the chain. The forward and backward definite functions are applied to these additional constraints to create another chain of states. The new chain branches from the state to which the constraints were added and ends in either a safety-critical deviation or a dead-end.

Further constraints can be added to the new state chains. The analysis procedure continues in a breadth-first manner, building a tree of state chains, each ending in either a dead-end or hazardous deviation. The depth of each leaf of the tree corresponds to the number of additional assumptions the procedure made to reach that leaf. The procedure finishes when it either runs out of constraints that it can make or the depth reaches some predefined limit set by the analyst. Finally, the procedure provides the analyst with scenarios by tracing each path from the initial state to all ending states that contain hazardous deviations.

SDA has been applied to three real-world examples. It was first applied to the Traffic Alert and Collision Avoidance System II (TCAS II), an avionics system designed to provide pilots with collision avoidance information. The authors found that when an incorrect identifier is received by TCAS (perhaps as a result of a transmission error) there are circumstances in which an evasive maneuver is not displayed to the pilot when it should be. The procedure has also been applied to a developmental aircraft guidance system and a proposed automated highway system with similar results. The time required to perform a search on these examples ranged from about 10 seconds to several minutes.

7 Example of the SDA Procedure

The basic procedure can be illustrated by a simple example from an actual project. First, the analyst provides a formal specification, which in this case is a proposed automated highway system for the California Department of Transportation[2]. The automated highway system (AHS) directs automobiles to form groups within a lane, called *platoons*. Each automobile has a software controller that directs the movement of the car relative to the platoons. Figure 5 shows a page from the AHS specification, written in Requirements State Machine Language (RSML) [1]. The relevant parts will be explained shortly.

Next, the analyst identifies the safety-critical outputs. For simplicity, in this example we will assume that all outputs are critical. The next step is to define the initial input deviations. One of the input variables listed at the top of the AHS specification is `Num_vehicles_in_platoon`, which is the number of automobiles that are in the same platoon as the controller's automobile. Suppose the analyst wishes to find out what happens when this input variable is less than the actual size of the platoon.

For this model, the SDA algorithm identifies two scenarios in which the input deviations lead to a safety-critical output deviation, one of which is presented here.[3]

The initial assumption, as stated above, is that `Num_vehicles_in_platoon` is too

[2]This research was supported by the Department of Transportation, and thus the models we have analyzed so far are not related to the chemical process industry. However, the technique is generally applicable to any control system software.

[3]The search for this example takes approximately 1.2 MB and 25 seconds on an Intel 80486DX2 at 66MHz.

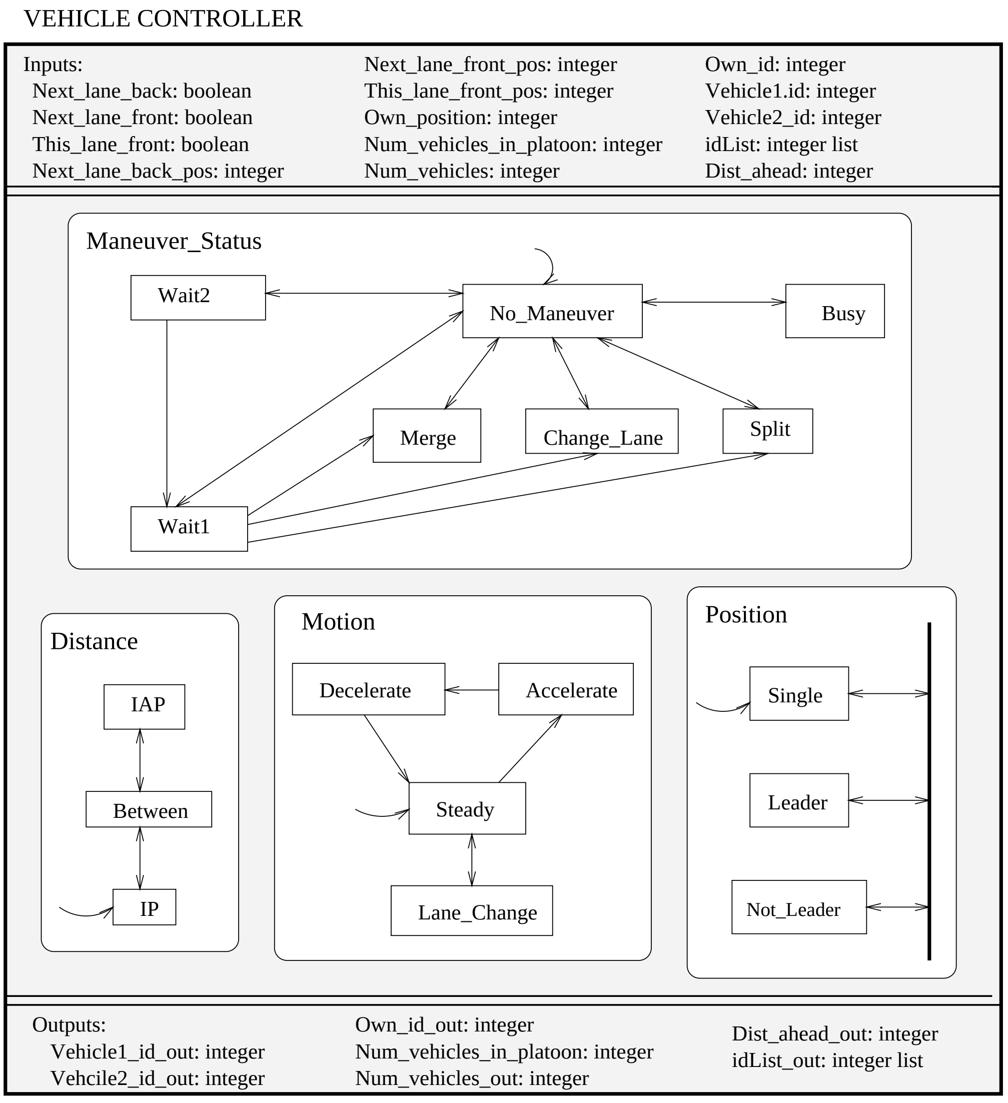


Figure 5: A portion of the Automated Highway System example.

Transition(s): $\boxed{\text{No\_Maneuver}} \longrightarrow \boxed{\text{Busy}}$

Location: Maneuver_Status

Trigger Event: Rcvd_req_merge

Condition:

		OR	
	Position **in state** Single	T	·
A	Position **in state** Leader	·	T
N	Distance **in state** IP	T	T
D	Motion **in state** Steady	T	T
	Num_vehicles + Num_vehicles_in_platoon < MAX_VEHICLES_IN_PLATOON	T	T

Output Action: Merge_ok

Figure 6: A transition from the Automated Highway System. The analyst has assumed that the variable `Num_vehicles_in_platoon` is lower than the actual value.

low. This input is used by the automobile's controller to determine whether to commence a maneuver, specified by the transition from `No_Maneuver` to `Busy` (refer to Figure 6.) This transition is taken if the controller has been requested to merge two platoons (the triggering event) and the guarding condition on the transition permits the platoons to merge.[4] For this scenario, the SDA algorithm constrains the software execution state in order to propagate the initial deviation through the transition, causing the controller to enter `Busy` when it should not (a boolean deviation.)

The input variable `Num_vehicles` is the number of vehicles wanting to merge with the platoon and the first constraint on the software state that SDA makes is that `Num_vehicles` is not too high, i.e., the value received is either the same as or less than the proper value. This constraint ensures that the sum of the two inputs in the fifth row of the table is too low. The deviation could be "masked" if `Num_vehicles` were too high. The second constraint that the algorithm makes is that the fifth row is true, namely, the maximum number of vehicles has not been exceeded based on the information provided to the software. The third constraint is that the sum of deviations for `Num_vehicles_in_platoon` and `Num_vehicles` exceeds the number of empty positions left in the platoon. In other words, the fifth row should be false, which it would be if the deviations were not present. The algorithm presents this logic

[4]The condition is represented by an AND/OR table, which is true if any of its columns is true. A column is true if all of the rows that have a "T" are true and all of the rows with an "F" are false.

in the following way (rearranged slightly for clarity):

$$\begin{array}{c}\texttt{MAX\_VEHICLES\_IN\_PLATOON}\\ -\texttt{value(Num\_vehicles)}\\ -\texttt{value(Num\_vehicles\_in\_platoon)}\end{array} < \begin{array}{c}-\texttt{dev(Num\_vehicles)}\\ -\texttt{dev(Num\_vehicles\_in\_platoon)}\end{array}$$

where `value()` is the value read by the controller and `dev()` is the difference between the actual and correct values (negative means too low.) The left-hand side of the inequality is the number of spaces available according to the two inputs read by the controller (`Num_vehicles` and `Num_vehicles_in_platoon`.) The right-hand side of the inequality is the size of the error (the deviation is negative, so the right-hand side is positive.) The inequality therefore shows that the deviation is greater than the number of spaces perceived to be available, and row five would be false if there had been no deviation in the inputs, inhibiting the transition.

The fourth constraint that the algorithm generates is that (1) the controller's automobile is the lead vehicle in the platoon, (2) the distance between the platoons is sufficient, and (3) the automobile is traveling at a constant rate of speed. These are the conditions in rows two through four, respectively, making the second column and the entire condition true.

The final constraint generated by the algorithm is that the triggering event is true, i.e., that a request has been made to merge platoons and the controller is in state `No_Maneuver`. The transition is thus enabled and the output action `Merge_ok` is generated. Both the transition and output action are deviations since they should not have occurred. `Merge_ok` triggers the output that gives permission to merge platoons. The resulting system state is a platoon that exceeds the threshold for safe platoon size. Thus we have identified a scenario, i.e., deviations in one input plus constraints on the software execution state that will lead to a hazardous output.

8 Conclusion

This paper has described a hazard analysis technique, software deviation analysis, that incorporates the beneficial features of HAZOP (e.g., guide words, deviations, exploratory analysis, and a systems engineering strategy) into an automated procedure that is capable of handling the complexity and logical nature of computer software.

We have applied SDA to several realistic systems. Although more extensive experimentation needs to be performed, the procedure appears to provide information that is useful for requirements specification and review. Note that SDA is not intended to replace standard certification methods such as verification and validation. However, as an exploratory procedure, it provides important and timely information to the safety and software analysts.

References

[1] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, SE-20(9), September 1994.

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

[3] J. D. Reese. *Software Deviation Analysis.* PhD thesis, University of California, Irvine, 1996.

[4] J. Suokas. The role of safety analysis in accident prevention. *Accident Analysis and Prevention*, 20(1):67–85, 1988.