Software Deviation Analysis

Jon Damon Reese
Dept. of C.S.E.
University of Washington
Box 352350
Seattle, WA 98195, U.S.A.
+1 (206) 616-1844
jdreese@cs.washington.edu

Nancy G. Leveson
Dept. of C.S.E.
University of Washington
Box 352350
Seattle, WA 98195, U.S.A.
+1 (206) 685-1934
leveson@cs.washington.edu

ABSTRACT
Validation of software requirements is an important part of software engineering. This paper describes a new safety analysis technique called *software deviation analysis* to help identify weaknesses in how software handles an imperfect environment. The technique propagates deviations in software inputs to output deviations. A qualitative analysis is used to improve the search efficiency.

Keywords
software safety, hazard analysis, software deviation analysis

INTRODUCTION

Because a large portion of software-related accidents arise from errors or omissions in the software requirements specification, validation of the safety of the specification is an important goal, especially if it can be done early in the development process. Although system safety engineers have developed various types of hazard analysis techniques for electro-mechanical 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 software and the software development and validation process.

This paper presents a new technique called *software deviation analysis* (SDA) [11], which is based on the underlying systems theory that accidents are caused by deviations in system parameters. Using a formal software or system requirements 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.

SDA is based on the concept of a *deviation*. A deviation is the difference between the actual value of a system variable and the expected (or "correct") value. Examples of deviations are "too high" and "too low." SDA analyzes the effect of deviations in software inputs on software output.

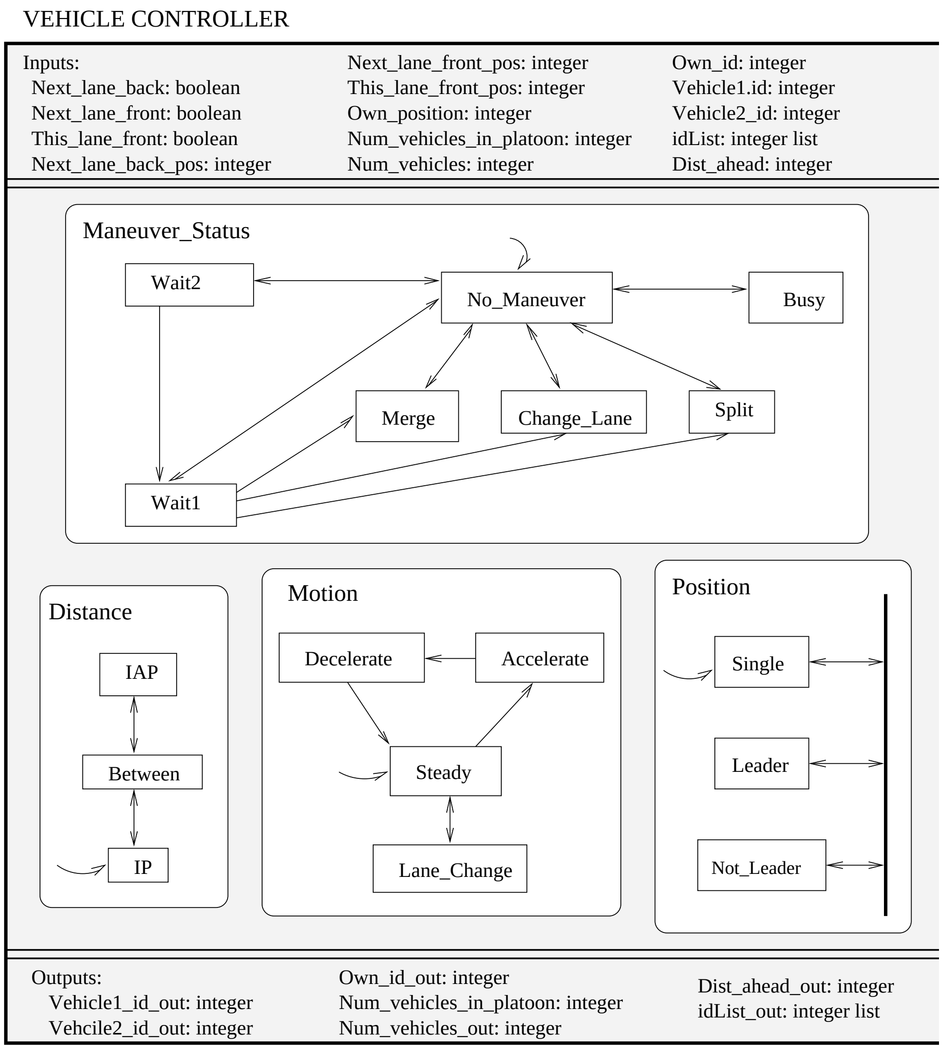
The basic inputs to the SDA algorithm are a formal software requirements specification, a list of safety-critical software outputs, and some initial assumptions about the values of the software inputs, including at least one deviation in these inputs.

The output of the SDA algorithm 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 an output deviation in a significant variable.

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. 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 1(a) shows a page from the AHS specification, written in Requirements State Machine Language (RSML) [5]. 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 scenar-



(a)

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

Location: Maneuver_Status

Trigger Event: Rcvd_req_merge

Condition:

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

Output Action: Merge_ok

(b)

Figure 1: (a) A portion of the Automated Highway System Example. (b) A transition from the Automated Highway System. The analyst has assumed that the variable `Num_vehicles_in_platoon` is lower than the actual value.

ios in which the input deviations lead to a safety-critical output deviation, one of which is presented here.[1]

The initial assumption, as stated above, is that `Num_vehicles_in_platoon` is too low. One of the transitions that references this input is the transition from `No_Maneuver` to `Busy` (refer to Figure 1.) 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.[2] 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` is 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 in the following way (rearranged slightly for clarity):

```
MAX_VEHICLES_IN_PLATOON − value(Num_vehicles)
      − value(Num_vehicles_in_platoon) <
−dev(Num_vehicles) − dev(Num_vehicles_in_platoon)
```

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.

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

[2]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.

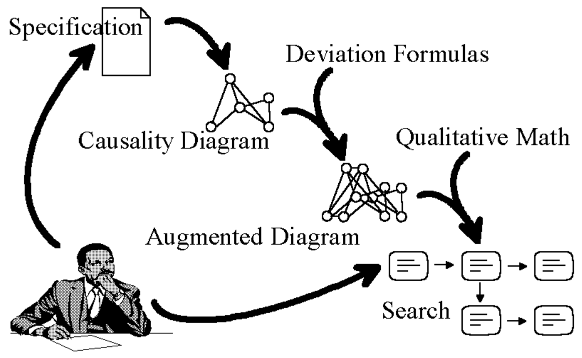


Figure 2: Overview of SDA procedure.

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.

OVERVIEW OF THE SDA ALGORITHM

Figure 2 shows an overview of the SDA process. 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.

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.

The 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 uses qualitative mathematics on the augmented causality diagram to evaluate deviations. Qualitative mathematics partitions infinite domains into small sets 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 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.

The rest of the paper provides more details about each part of this process along with a brief description of the automated procedure and the results we have had in applying SDA to specifications.

CAUSALITY DIAGRAMS

Rather than being defined in terms of a particular specification language, the search procedure is based on causality diagrams, which encode causal information between system variables. Although a different procedure could be developed for each separate specification language, there are considerable advantages in developing a procedure around a simpler language of causality. Development of the analysis procedure is simplified if the semantics of the specification language are straightforward, simple, and explicit. A second advantage is that the procedure can be divided into two steps: (1) translating the relevant parts of the language into a more fundamental representation, and (2) developing a procedure for the simpler language. Finally and perhaps most important, this tactic assists in adapting the procedure for multiple languages: It is easier to build generators of causality diagrams from multiple specification languages than to design analysis procedures that work on multiple, often subtly different languages.

A potential disadvantage of using a separate analysis language is that the analyst may need to 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 in order to provide input to the procedure. In addition, a variable in the analysis model always maps back to a single variable or a set of equivalent variables in the specification. Thus the procedure's results (including a full search tree) can be satisfactorily presented to the analyst in terms of the specification model. The analyst does not need to inspect the causality diagram at all.

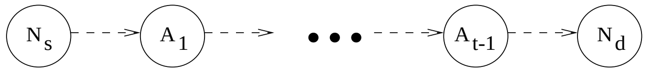
To be suitable for automated analysis, a language of causality must represent relationships directly (i.e, there should be no intermediate variables or "steps" that take no time), must be able to represent complex relationships, must represent sequential dependency and simultaneity explicitly rather than relying on relationships implied by the language's semantics, and must be able to handle both boolean and numeric system variables.

A causality diagram is a type of graph, i.e., 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 meaning. Some functions are not commutative, so the edges into the nodes must be ordered. The values of the nodes comprise the state of the causality diagram.

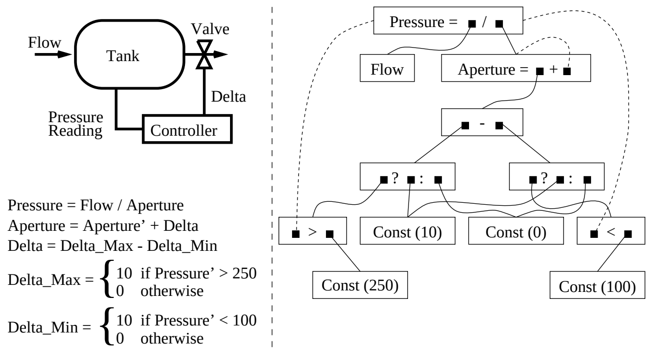
Nodes in a causality diagram are divided into *source nodes* and *auxiliary nodes*. The source nodes correspond to the system variables. Auxiliary nodes are used to compose functions of source nodes, to propagate deviations for functions of fixed arity, and to represent unspecified input sources and output destinations.

Each node is associated with an algebraic or logical operator. The operator describes the exact relationship between a node and nodes from which it derives its value, i.e., the causal relationships. The edges into the node give two pieces of information. The edge's source node is the node that contributes its value to the destination node's operator. Each edge also describes either a *structural* or *sequential* relationship, depending on whether the source node contributes its value in the same instant or in the following instant. In the figures presented in this paper, structural edges are represented as solid lines and sequential edges are represented as dashed lines.

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



A node may be both source and terminus for a sequential edge. 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.



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

Putting the concepts of structural and sequential causality together, Figure 3 shows an example of a simple feedback system and the corresponding causality diagram. In a feedback control loop, sensors measure system variables and pass them to the controller, which uses them along with an internal model of how the process operates (e.g., control laws) to generate commands to actuators (such as valves). The changes in the actuators result in changes to system variables that allow the process to produce the required results. The controller uses the feedback provided by the measured system variables to determine whether previous commands were effective and to detect disturbances in the process that need to be corrected. 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. 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 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 remaining nodes comprise the controller (which for process-control software would be a computer, usually performing a much more complex function.) The pressure 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.

A full discussion of the translation of specification language concepts into a causality diagram is beyond the scope of this paper. Reese [11] has shown how to do this in general and has also written a translator from RSML specifications to causality diagrams. The size of the causality diagram is proportional to the number of system variables in the specification and the number of mathematical operators needed to describe their basic relationships. The causality diagrams in our non-trivial test cases have between 5,000 and 20,000 nodes.

We show here the translation of an RSML transition using the specification of an aircraft collision avoidance system (TCAS II) as an example.

RSML transitions have five components:

- a source state (S),
- a destination state (D),
- a triggering event (E),
- an optional guarding condition (C), and
- an optional output action (A).

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

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

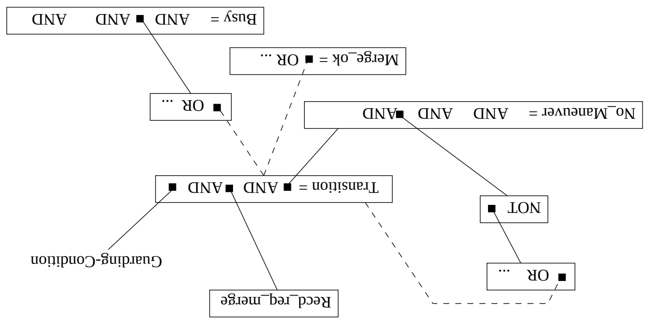


Figure 4: A causality diagram fragment for the transition from *No_Maneuver* to *Busy* (see figure 1b.)

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 1b is shown in Figure 4.

A causality diagram makes the definition of each system variable explicit. These definitions describe the normal behavior of the system. The next step is to use deviation formulas to add the causal relationships between deviations.

DEVIATION FORMULAS AND AUGMENTED DIAGRAMS

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. $X \oplus Y$ is *true* when X is different from Y 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.[3] 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, $\texttt{Pressure}_a = \texttt{Flow}_a / \texttt{Aperture}_a$.

[3]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 $X_a = 0$.

Deviation values cannot be calculated using the causality diagram as it is: $\texttt{Pressure}_d$ is not always equal to $\texttt{Flow}_d / \texttt{Aperture}_d$. (In fact, if the actual value of $\texttt{Aperture}$ 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.

Both the boolean and numeric deviation definitions have two degrees of freedom, i.e., knowing any two of the three variables (correct value, actual value, and deviation) allows calculation of the third. Thus, correct values can be replaced by a function of the actual and deviation values, as the following example for multiplication shows:

$$\begin{aligned}
(XY)_d &= (XY)_a - (XY)_c \\
&= X_aY_a - X_cY_c \\
&= X_aY_a - (X_a - X_d)(Y_a - Y_d) \\
&= X_aY_a - (X_aY_a - X_dY_a - X_aY_d + X_dY_d) \\
&= X_dY_a + X_aY_d - X_dY_d
\end{aligned}$$

Replacing the correct value with its computation using actual values and deviations has two advantages: it simplifies the analysis because fewer values need to be calculated, and it simplifies the interpretation of the results because the analyst is only presented with what actually occurs rather than mixing what should occur with what does occur.

A complete set of deviation formulas and their derivations can be found in [11]. 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 5.

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 $\mathcal{D}$ 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 introduced qualitative mathematics is most commonly referred to as "qualitative reasoning," although variations in research emphasis have

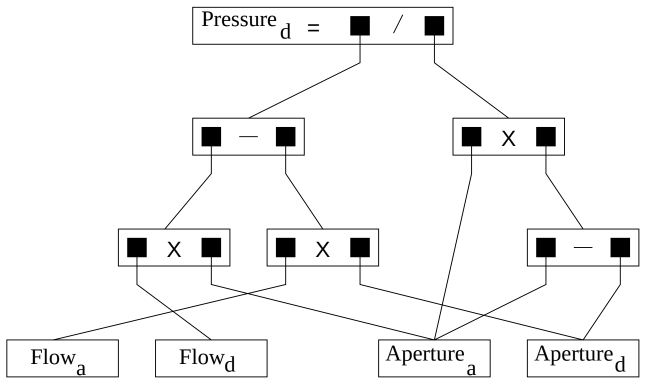


Figure 5: A fragment of the augmented causality diagram for the example, showing the causal relationship between deviation in pressure and the deviations and actual values of flow and valve aperture.

$x + y$

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

$x \times y$

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

Table 1: Sign algebra.

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. SDA 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 $N = f(I_1, ...)$, where f is the node's function and $I_1, ...$ are input nodes. An analysis procedure may apply the calculus to the set of system axioms to produce "theories" of system behavior.

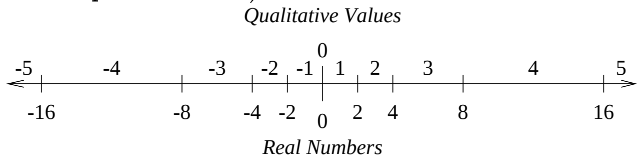
A simple and commonly used qualitative domain is the set of signs of the real numbers, $S_Q = \{-, 0, +, ?\}$. S_Q partitions the real numbers into two sets, the positive $(+)$ and negative $(-)$ numbers. Zero (0) is the border between the two sets. The special symbol "?" represents an unknown value and is equivalent to the union of the other symbols. The algebraic functions over S_Q are referred to as *sign algebra* in the literature. Addition and multiplication are shown in Table 1. Qualitative mathematics is not limited to algebraic functions. For example, Schaefer constructs an interesting model for a family of nonlinear oscillating functions [12].

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.

Some operations over qualitative sets do not fall neatly within one of the intervals. For example, note that the sign algebra tables contain question marks in some cells. Increasing the number of elements in the qualitative set alleviates this problem to some degree. For example, one could add two elements representing the fraction intervals $(-1, 0)$ and $(0, 1)$.

Another potential weakness of qualitative mathematics is that of scale. Specifications involve both large and small numbers and the qualitative set should be able to handle both. We use a logarithmic partitioning of the real numbers to solve this problem. For example, the following number line is divided into eleven intervals (including the zero point interval), with a base of 2 (again, Reese [11] provides a set of proofs of the calculus of this qualitative set):



The analyst may choose to refer to the qualitative set elements, the intervals they represent, or to symbolic labels such as "slightly too high." The HAZOP guide word TOO-HIGH is represented by the qualitative subset $\{1, 2, 3, 4, 5\}$, which is $(0, +\infty)$. If the analyst chooses to assume that an input is a "little low" then -1 may be chosen, which is the interval $[-2, 0)$ for the example given above. "Very high" could be represented by the qualitative value 5, which is all numbers greater than 16. Of course, choosing a larger number of elements and a larger base allows a greater partitioning of the real numbers. Refer to [11] for a complete description of the logarithmic partitioning strategy.

In a manual hazard analysis, 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. SDA also needs to include these three types of search, i.e., to propagate deviations forward. The mathematical equivalent to the informal activity is termed the *assumptive function.* Each operator has three functions defined: *forward definite*, *forward assumptive*, and *backward definite.*

The forward definite function is simply the operator itself, but applied to sets of intervals. For example, $\{0,+\}/\{-\} = \{-,0\}$ under sign algebra. 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 ($\{-,0,+\}$). In order to propagate the high value, the unknown input is assumed to be $\{-,+\}$, since 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 [11] for definitions of these relations.

ANALYSIS PROCEDURE

The SDA procedure is a forward search procedure—it starts with a deviation in software inputs prior to being input to the software and attempts to find ways in which the deviation can lead to hazardous software outputs. As discussed in the overview, the analysis uses a system specification that is then converted into an augmented 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 can follow. 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 combined with the algorithm's constraints on the software state 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.

New constraints can be added to states in 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 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.

The number of possible states is exponential in the number of nodes. The search tree can in the worst-case contain each state in the state space, so it can also grow exponentially to the number of nodes. The algorithm maintains the visited states in a hash table so the size of the search tree can grow no larger than this theoretical maximum.

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 escape maneuvers from intruding aircraft. 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 escape 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 ranged from about 10 seconds to approximately 20 minutes. The search space for these examples ranged from a single state (e.g., the initial state contained a significant deviation) to several thousand states.

RELATED METHODS

The type of hazard analysis closest to software deviation analysis is called HAZard and OPerability analysis (HAZOP). HAZOP is a review procedure developed for the British chemical industry in the 1950's to cope with potential hazards and other disturbances in operations.

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

Table 2: HAZOP guide words (adapted from Leveson [6].)

The goal of a HAZOP is to identify operational deviations from intended performance and study their impact on system safety [13]. 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 [8]. 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 is based on a systems theory model of accidents [6], 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 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 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?"

HAZOP has several limitations. First, it is time- and labor-intensive [6], 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 [13]. A purely flow-oriented approach may cause the analyst to neglect process-related malfunctions and hazards in favor of pipe-related causes and effects.

Component	Guide Word
Whole machine	gross failure
Misc. parts	random failure
Signal	low, high, invariant, drifting, bad
Actuator	driven/failure high, driven/failure low, stuck, drifting

Table 3: Components and guide words suggested by Andow [1].

OMISSION	Intended output missing.
COMMISSION	Unintended output.
EARLY	Output occurs too soon.
LATE	Output occurs too late.
COARSE INCORRECT	Output's value is wrong.
SUBTLE INCORRECT	Output's value is wrong, but cannot be detected.

Table 4: Computer HAZOP guide words suggested by McDermid and Pumfrey [7].

Because HAZOP concentrates on physical properties of the system [13], 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 may differ from the original pipe-and-process diagram. Andow [1], for example, proposes augmenting a standard HAZOP with the components and guide words listed in Table 3. Burns and Pitblado [2] propose applying the guide words "no," "more," "less," and "wrong" to all computer inputs and outputs.

McDermid and Pumfrey [8] suggest applying a different list of guide words (see Table 4) using a data-flow diagram of the software design specified in a language called MASCOT. They apply their guide words to software outputs *only* so their procedure is really closer to a Failure Modes and Effects Analysis (FMEA) than a HAZOP, where the six "guide words" are used as generic modes of software failure. No evaluation is made of whether the components could actually fail in this way given their specified functionality but the potential effects of such failure modes are evaluated.

One problem with developing an automated technique based on a MASCOT specification is that only data flow (or "information flow" in the authors' terminology) can be analyzed. While this strategy is faithful to the standard HAZOP procedure, it precludes an analysis based on deviations in system components other than data paths. This weakness does not theoretically 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. However, without looking at the defined or required functionality of the software itself, the analysis is limited in the type of information that can be obtained. A primary goal of software hazard analysis is to identify weaknesses in the specified software functionality, and an analysis that stops at the border of each component does not provide the necessary detail to find this type of problem.

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.

The generality of the guide-word "COARSE INCORRECT" also leads to problems. This single guide word replaces several standard HAZOP guide words, such as "HIGH" and "LOW" and leads to loss of what can be important information.

Other manual techniques are variants of the ones described [4, 9, 10, 3]. All of 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.

In contrast to these manual techniques that try to do no more than HAZOP (which was originally designed for analog systems), software deviation analysis is an automated procedure that can trace the effects of deviations in component inputs to their effects on the outputs and eventually to important system parameters. The analysis can be applied to any system or software requirements specification that defines the required relationship (function) between inputs and outputs and thus can be used early in system development to identify hazards so they can be eliminated or controlled.

Digraph methods use a sort of qualitative calculus to analyze flow deviations. However, digraph 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 for SDA is specifically a calculus of deviations: The deviation values characterize the a parameter's actual value in relation to its correct value.

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 is the boundary between small and large values. $(L - 1)$ is a small positive, and the sum of two small positives is a small positive under digraph algebra. But $(L - 1) + (L - 1) = 2L - 2$ is large for $L \geq 2$, 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.

CONCLUSION

As software assumes a larger role in the control of safety-critical systems, a greater burden is being put on analysts to understand how the software behaves in an imperfect environment. Correct and complete software requirements specifications must define behavior that can handle environmental disturbances without contributing to system hazards. Although forward search procedures, such as HAZOP, have been found to be useful in the hazard analysis of systems made up of analog components, it needs to be extended to be appropriate for systems that contain digital components. This paper describes a requirements validation method 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.

SDA may be thought of as a type of symbolic execution: 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 provides derivations of the conditions that can lead from an initial state to a hazardous state. As discussed, this derivation may require adding software state constraints to the input deviations. The steps of the derivation are composed using rules of the calculus (the qualitative functions) and the axioms (the relations in the causality diagram).

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 information to the software analyst.

REFERENCES

[1] P. Andow. Guidance on HAZOP procedures for computer-controlled plants. Her Majesty's Stationery Office, London, 1991.

[2] D. Burns and R. Pitblado. A modified HAZOP methodology for safety-critical system assessment. In F. Redmill and T. Anderson, editors, *Directions in Safety-Critical Systems: Proceedings of the Safety-Critical Systems Symposium*, London, 1993. Springer-Verlag.

[3] R. Fink, S. Oppert, P. Collinson, G. Cooke, S. Dhanjal, H. Lesan, and R. Shaw. Data management in clinical laboratory information systems. In F. Redmill and T. Anderson, editors, *Directions in Safety-Critical Systems: Proceedings of the Safety-Critical Systems Symposium*, London, 1993. Springer-Verlag.

[4] J. Lear. Computer hazard and operability studies. In *Sydney University Chemical Engineering Association Symposium: Safety and Reliability of Process Control Systems*, October 1993.

[5] 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.

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

[7] J. McDermid. Issues in developing software for safety critical systems. *Reliability Engineering and System Safety*, 32(1–2):1–24, 1991.

[8] J. McDermid and D. Pumfrey. A development of hazard analysis to aid software design. In *COMPASS*, 1994.

[9] Draft interim defence standard 00-58: A guideline for HAZOP studies on systems which include a programmable electronic system. Ministry of Defence, Directorate of Standardization, Glasgow, UK, 1995.

[10] I. Nimmo, S. Nunns, and B. Eddershaw. Insert title here. *Loss Prevention Bulletin*, 111:13, 1993.

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

[12] P. Schaefer. Analytic solution of qualitative differential equations. In *Proceedings Ninth National Conference on Artificial Intelligence*, pages 830–835. AAAI Press, July 1991.

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