Deviation Analysis Through Model Checking[*]

Mats P.E. Heimdahl, Yunja Choi, Mike Whalen
Department of Computer Science and Engineering, University of Minnesota
200 Union Street S.E., 4-192, Minneapolis, MN 55455, USA
{heimdahl,yuchoi,whalen}@cs.umn.edu

Abstract

Inaccuracies, or deviations, in the measurements of monitored variables in a control system are facts of life that control software must accommodate—the software is expected to continue functioning correctly in the face of an expected range of deviations in the inputs. Deviation analysis can be used to determine how a software specification will behave in the face of such deviations in data from the environment. The idea is to describe the correct values of an environmental quantity, along with a range of potential deviations, and then determine the effects on the outputs of the system. The analyst can then check whether the behavior of the software is acceptable with respect to these deviations.

In this report we wish to propose a new approach to deviation analysis using model checking techniques. This approach allows for more precise analysis than previous techniques, and refocuses deviation analysis from an exploratory analysis to a verification task, allowing us to investigate a different range of questions regarding a system's response to deviations.

1. Introduction

Often, because of inherent limitations in sensors, a control system is presented with slightly inaccurate information about its environment. These inaccuracies are deviations from the actual value of an environmental variable. These deviations can stem from a number of sources: inaccurate sensors, electrical interference on a wire, a garbled message over a bus, etc. Frequently, control software must continue to function correctly within an expected range of deviations in the inputs.

Deviation analysis is concerned with discovering and classifying any changes in system behavior between two identical control systems in slightly different environments. One system is provided with absolutely accurate input data from the environment, and the other is provided with a slightly inaccurate *deviation model* of the environment, created by an analyst. Deviation analysis can be distinguished from standard verification and validation activities in that the control system in question, given correct inputs, is always assumed to be correct—we are not interested in looking for faults in the model under perfect operating condition. Instead, it is a mechanism for determining the robustness of the control system in the face of expected inaccuracies in input data.

This idea has been explored in previous work [9, 10, 11] using symbolic execution and partial evaluation. Given qualitative descriptions (e.g. "very high", "low") of deviations on system inputs, this work allows open-ended exploratory analysis of effects on system outputs. For example, using the symbolic execution technique, one can ask: *"What are the possible effects on the primary flight display if the altitude reading is deviated so that it is 'higher' than the correct value?"*. Unfortunately, the qualitative abstractions used in this approach lead to situations where it is difficult to determine how a deviation will affect the value of an output.

In this paper, we describe an alternate approach to deviation analysis. Our approach works by restating exploratory deviation analysis questions as verification tasks suitable for model checking [3]. For example, the flight display question can be restated as follows: *"Will the correct and deviated primary flight displays always match if the altitude reading is high by 0 to 100 feet?"*. This approach is more precise than previous approaches and supports an alternate verification style. We have defined the mechanism for performing this analysis and implemented a prototype tool.

In the next section we describe previous approaches to deviation analysis in more detail. Sections 3 and 4 describe how deviation analysis questions are formulated as verification problems and implemented using a model checker. Section 5 presents a small example of the approach. Finally, Section 6 provides a short discussion and future directions.

[*]This work has been partially supported by NASA grant NAG-1-224 and NASA contract NCC-01-001.

2. Background

Reese and Leveson introduced the notion of *software deviation analysis* in 1996 [11]. The method is based on the Hazard and Operability (HAZOP) analysis [2, 5], a successful procedure used in the chemical and nuclear industries. This section gives the reader an overview of the evolution of software deviation analysis and the developments that led to the approach described in this paper.

Hazard and Operability analysis (HAZOP): Developed for the British chemical industry in the 1950's, HAZard and OPerability analysis (HAZOP) [2, 5] is a manual analysis technique in which a HAZOP leader directs a group of domain experts to consider a list of *deviations*. In HAZOP, a deviation is the combination of a *guide word*, such as "too-high," with a system variable, *e.g.*, "expansion tank pressure," yielding a question: *"What is the potential effect of the pressure in the expansion tank being too high?"* The answer to the question is now used to pose additional questions (following the same "guide-word/system variable" approach) about the effect of the result of the first question. In this way, the deviations are propagated through the system in an attempt to discover hazardous results.

There have been some attempts to adapt the manual HAZOP technique to include software [6], but these techniques are 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 plant diagrams from the chemical processing industry (pipes, tanks, and valves) used in the original approach. Because of the complexity of control software (as compared to the pipe diagrams of the past) and the lack of a formalism for propagating the deviations through the software, these techniques are largely infeasible in practice. To address these problems and bring HAZOP related techniques to the safety critical software field, Reese and Leveson developed *software deviation analysis*.

Software Deviation Analysis: Software Deviation Analysis [9, 10, 11] overcomes some deficiencies with HAZOP. Software Deviation Analysis is based on the same underlying idea as HAZOP—accidents are caused by deviations in system parameters. Using a blackbox software or system requirements specification, the analyst provides assumptions about particular deviations in software inputs and hazardous states or outputs, and the software deviation analysis automatically generates scenarios in which the analyst's assumptions lead to deviations in the specified outputs. 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; in a sense, deviation analysis is a symbolic execution of a software requirements model including the deviations in the evaluation.

To represent the deviations, Reese and Leveson use qualitative mathematics—a branch of mathematics that operates on categories of numbers rather than the numbers themselves. For instance, a negative number multiplied by a negative number equals a positive number. The advantage to deviation analysis is that general results (whole classes of deviations) can be propagated quickly and clearly. In [11], Reese developed a calculus of deviation that formed the basis for their tools—a sample expression from the calculus is "Very High-Positive $\times$ Normal-Negative $=$ Very Low-Negative." With this calculus and the deviation analysis tools, an analyst can pose a questions such as "If the altitude altitude reading on altimeter 1 is 'Very-High-Positive', what will be the effect on the pitch command?"

In an attempt to implement this analysis procedure in the NIMBUS tool at the University of Minnesota[1], we came to the conclusion that the qualitative mathematics used in software deviation analysis did not provide the analysis accuracy that we required. Therefore, we developed a related approach based on interval calculus as opposed to qualitative mathematics.

Perturbation Analysis: Perturbation analysis is an adaptation and simplification of Reese's deviation procedure for the RSML^{-e} language. RSML^{-e} [12, 13] is a synchronous dataflow language in which the specification state is comprised of a set of *state variables*, each describing a portion of the specification behavior.

In perturbation analysis, the user specifies a state variable of interest (VOI), and we construct a partial machine state containing perturbed values of input variables that we use to compute the nominal and perturbed values of the VOI. We can then study these values to determine if the perturbation is within acceptable limits.

For example, suppose we have a state variable z, defined as follows (using the textual RSML^{-e} syntax):

```
 STATE_VARIABLE z: INTEGER
   PARENT : NONE
   INITIAL_VALUE : 0
   CLASSIFICATION : State

   EQUALS x * 2 IF b
   EQUALS y IF not (b or c)
   EQUALS x * 2 - y IF not b and c
END STATE_VARIABLE
```

From the definition, z references two input variables, x and y. Perturbations in x and y will manifest themselves in perturbations of z. Therefore, we must prompt the user

[1]NIMBUS is a execution, analysis, and code generation environment for the sate-based, fully formal specification language RSML^{-e}.

to decide to what extent *x* and *y* are perturbed. The user provides a range of correct values and also a range of perturbation.

For our example, a user could define *x* and *y* as follows:

Variable	Correct Range	Perturbation	Range
x	[0..10]	[-1..3]	[-1..13]
y	[5..60]	[10..20]	[15..80]

In this scenario, the correct value of *x* could range from [0..10], but because of sensor errors, each value of *x* could be perturbed anywhere from 1 less than its true value to 3 greater than its true value. Thus, the potential range of perturbed values is from [-1..13]. We compute both the potential correct values of the VOI and a range of perturbed values given perturbed inputs.

Because we are describing variable ranges, several different assignment conditions could hold for a given state variable. Therefore, we output of the analysis results a set of ⟨ *condition, correct, perturbed* ⟩ tuples, where if condition *condition* holds, then *correct* describes the possible correct range of values and *perturbed* describes the maximum perturbation possible.

In our example, given perturbations in *x* and *y*, the tuples are as follows:

Condition	Correct *z* Range	Deviated *z* Range
b	$[0..20]$	$[-2..26]$
$\neg(b \vee c)$	$[5..60]$	$[15..80]$
$\neg b \wedge c$	$[-60..15]$	$[-82..11]$

This analysis is appealing since it helps answer many questions during safety analysis that can be very difficult to address without tool support.

Issues with Existing Approaches: These analyses, while useful, have issues that must be resolved before they are applicable in realistic applications. The most serious issue involves the interplay between intervals (whether numeric or qualitative) and Boolean conditions within the specification. The intervals are often too large to accurately partition the conditions into cases in which the deviated specification behaves differently than the non-deviated specification. For example, given the definition of z, a useful question might be: *"Are there any circumstances when the non-deviated z is greater than 18 but the deviated value is less than 18?"* For this question, the output of the deviation analysis as well as perturbation analysis provides no help. The user must go back and describe smaller correct intervals in order to improve the accuracy of the analysis.

Also, given the interval procedure above, we can determine the maximum and minimum of the deviated *range*, but it is not as straightforward to determine the maximum and minimum *deviation*. Similarly, with qualitative methods, the output of the analysis describes only qualitative ranges of deviations.

Finally, the perturbation analysis was originally defined is a 'one-step' analysis; it does not record how a series of perturbations affects the specification over time. Unfortunately, many of the properties that one is interested in (e.g. stability) can only be checked over multiple steps. The deviation analysis, while allowing multiple steps, requires user guidance to successfully explore a multi-step state space. While pondering these drawbacks, the solutions started to look more and more like some variant of temporal logic model checking. This fact led to our investigation of alternative approaches to symbolic execution and was the genesis of the model checking ideas presented in the next section.

3. Deviation analysis as a verification problem

Because of the issues with the symbolic execution approach discussed above, we developed a novel technique to use model-checking techniques to perform deviation analysis[2]. As mentioned above, deviation analysis is intended to answer *"What if?"* questions such as *"What is the effect on the output DOI-Command if the altitude reading is 'high'?"*. This question can be explored with the techniques discussed in the previous section. By restating the question to *"Will there be an effect on the DOI-Command if the altitude reading is off by 0 to 100 feet?"*, we change the analysis from an exploratory analysis to a verification task suitable for model checking.

Consider the example with *x*, *y*, and *z* introduced above. In the original statement, we are simply interested in computing all variables that are data dependent on *x* and *y* in any way. We would then investigate the result and see if any of the affected variables had a deviation that was unacceptably large. If we restate our problem as *"Given a deviation of x and y, will the deviation of z be within an acceptable margin?"*, we can formulate it as a temporal logic property and, with creative use of a model checker, verify that the deviation is acceptable or provide an example (counter example) of how the deviation of *z* may become too large.

The general approach to deviation analysis using model checking is to represent the system under investigation with two models, one representing the behavior of the system with no deviation and the other representing the deviated system (Figure 1 outlines the general approach). We want the two models to operate on exactly the same inputs, except for the input variables that are deviated—we can then compare the computed states of the two systems and see if any critical deviations are present.

To assure that the discrepancy of computed states of the two systems are purely due to the given deviation, we im-

[2]We will use Reese's and Leveson's original name of the analysis since we abandoned perturbation analysis before it was fully implemented in a usable tool.

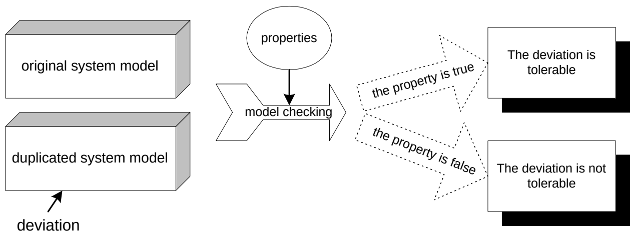

Figure 1. General approach of the deviation analysis using model checking.

pose two types of constraints on input variables; (1) for non-deviated input variables, both system models must receive the same values in each step, (2) for deviated input variables (only existing in the deviated system model), all deviated variables have exactly the value of the corresponding input variables in the original system plus possible deviations. The two system models are tied together through these constraints on the input variables.

As an example, let us use the *xyz* example from above. We would provide two system models; one expressed in terms of the variables *x*, *y*, and *z*, and the other (deviated system) in terms of the variables *x_d*, *y_d*, and *z_d*. Note that the models would be identical except for the names of the variables. We can now tie the 'correct' and 'deviated' system together with constraints on the input variables. Let us assume that we want to investigate how *z* is affected by a *[0..10]* deviation of *x*. By defining the constraints *y_d=y* and *x_d=x+[0..10]* we have stated that there is no deviation in the variable *y* and a deviation of *[0..10]* in variable *x*. If we want to investigate if *z* is affected by the deviation, we can state that it is globally true that *z=z_d*. If this property can be verified, the deviation in *x* does not propagate to *z*. If verification fails, we will get an example of a situation when the deviation in *x* shows up as a deviation in *z*. We can also capture the notion of acceptable deviations using this approach. For instance, if *x* is deviated, an acceptable deviation of *z* might be *e*. This can be stated as it is globally true that $(z\_d - e \leq z \leq z\_d + e)$.

The greatest challenge of using model checking for deviation analysis is in dealing with the state space explosion problem. Since we are duplicating the system model, the number of system variables may be doubled and, consequently, the size of the state space may explode. Also, since we are often interested in the actual *values* of data variables, dealing with data variables ranging over large domains is an issue that must be overcome. We will discuss these issues in Section 4 where we discuss the implementation of deviation analysis using a model checker.

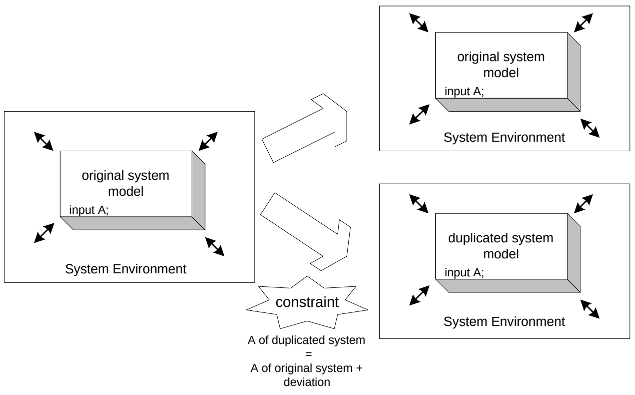

Figure 2. Modelling Scheme 1—simple duplication.

4. Deviation analysis using a model checker

The analysis ideas outlined above can be realized by either (1) providing two models of the original system and tie the input variables together with constraints, or (2) providing one model that is instantiated twice under the same system environment. We will discuss these approaches and their pros and cons in some detail below.

System model duplication: As shown in Figure 2, we can simply duplicate the original system model and check for critical discrepancies between the behavior of the original system and that of the duplicated system with deviation introduced.

The constraints between input variables of the original system and the duplicated system can be imposed outside of the two models as invariants. For example, suppose the original system has two input variables *X* and *Y* and we want to perform a deviation analysis on the input variable *X*. The constraints would be imposed in NuSMV [8] as follows (variables with a *d* subscript represents the variables in the deviated system):

INVAR Y_d = Y
INVAR X_d = X + deviation

The benefit of using this approach is mainly its simplicity; once we have an automated translation between a program or a specification, and a target input language of a model checker, no extra work is required for this approach other than duplicating the system and imposing constraints between the input variables. This approach, however, is inefficient since it duplicates all variables (and thus, increasing the state space when model checking) and it requires costly computations during model checking such as invariant assignments and computations.

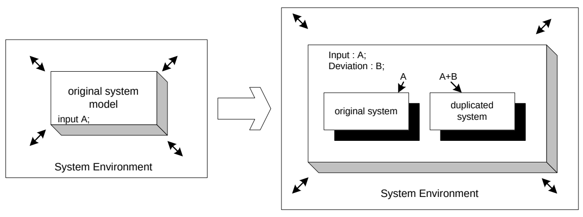

Figure 3. Modelling Scheme 2—embedding.

System model embedding: An alternative approach is to embed the original system and the deviated system inside of a common environment and check the outputs of both embedded systems. As shown in Figure 3, we use a two level model hierarchy. The top level is responsible for modelling the input variables to the system. The 'correct' and 'deviated' versions of the system models are represented as subsystems. The top level will then pass the necessary variables to the subsystems—variables that are not deviated will be passed to both subsystems as they are whereas variables that are deviated will be passed with a deviation added to the subsystem designated to be 'deviated'. The two subsystems execute synchronously and they compute the values of state variables based on the input values received from the parent. We can now express the properties of interest as properties over the two subsystems.

Note that the input variables are declared only once in the parent system in this modelling scheme and we do not need to impose extra constraints for input variables as invariants—this saves on both verification time and required memory space.

The table in Figure 4 shows a performance comparison of the two modelling schemes for one deviation analysis over the our sample system, the Altitude Switch, (after applying abstraction on numeric variables as described in the next section).

usage / scheme	# of BDD variables	memory usage	time usage
simple duplication	221	10 M	41.21 s
embedding	161	5 M	0.96 s

Figure 4. A comparison of the two modelling schemes.

Since the embedding scheme is more efficient, we have chosen to pursue this approach for our prototype tool. We have implemented a prototype tool that uses RSML^{-e} as the source language and produces output to NuSMV [8]; nevertheless, our approach is not limited to any particular model checking tools. In the next section we will illustrate the approach with a simple example.

5. A small example

We will use a very simple system from the avionics domain to illustrate our approach to deviation analysis. The Altitude Switch (ASW) is a (somewhat) hypothetical device that turns power on to another subsystem, the Device of Interest (DOI), when the aircraft descends below a threshold altitude and turns the power off again after we ascend over the threshold plus some hysteresis factor (the example is adopted from [7, 12]). The robustness to deviations in the altitude measures is the subject of interest in this section.

The ASW: The version of the ASW used in this paper receives altitude information from some number of radio altimeters. The functioning of the ASW can be inhibited or reset at any time. This raises questions, for example, about how the ASW should operate if it is reset or inhibited while crossing the various thresholds. In our initial version of the ASW, we model the perceived altitude status (are we above or below the thresholds) as shown in Figure 5. Using the textual RSML^{-e} syntax, we view the *AltitudeStatus* to be *Unknown* at system startup or after a reset. The variable is assigned the the value in an EQUALS clause when the guard condition in that clause is true. The guard condition is expressed in a tabular format we call AND/OR tables. The 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 match the truth values of the associated predicates. A dot denotes "don't care." For example, we will set the *AltitudeStatus* to *Above* or *Below* when we have determined that we are above the threshold hysteresis or below the threshold respectively—until then, we consider the *AltitudeStatus* to be *Unknown*. The *BelowThreshold()* and *AboveThresholdHyst()* macros encapsulate the conditions used to determine this information based on the altimeter data, this is where potential voting algorithms providing fault tolerance would be modelled.

The ASW command to the Device of Interest (DOI) is defined as in Figure 6. At startup and after a reset, we do not know what the to do with the DOI so we view its status as *Undefined*. We power the DOI off under two conditions, (1) we ascended above the threshold plus the hysteresis value (the condition *@T(..AltitudeStatus = Above)* indicating that the condition *AltitudeStatus = Above* became true) while not being inhibited nor reset, or (2) we are currently above the threshold plus hysteresis and the reset is removed. We turn on the DOI if the system is operational (i.e. it isn't inhibited or reset) and the altitude changed from above to below (condition rows 1 and 4).

```
STATE_VARIABLE AltitudeStatus :
   VALUES : {Unknown, Above, Below, AltitudeBad}
   PARENT : PowerStatus.On
   INITIAL_VALUE : UNDEFINED
   CLASSIFICATION : State

   EQUALS Unknown IF
     TABLE
      ivReset                                  : T *;
      PREV_STEP(..AltitudeStatus) = UNDEFINED  : * T;
     END TABLE

   EQUALS Below IF
      TABLE
         BelowThreshold()     : T;
         AltitudeQualityOK()  : T;
         ivReset              : F;
      END TABLE

   EQUALS Above IF
      TABLE
         AboveThresholdHyst() : T;
         AltitudeQualityOK()  : T;
         ivReset              : F;
      END TABLE

   EQUALS AltitudeBad IF
      TABLE
         AltitudeQualityOK()  : F;
         ivReset              : F;
      END TABLE
END STATE_VARIABLE
```

Figure 5. The definition of the *AltitudeStatus* **state variable.**

```
STATE_VARIABLE DOI_Intended :
   VALUES : {PowerOff, PowerOn}
   PARENT : PowerStatus.On
   INITIAL_VALUE : UNDEFINED
   CLASSIFICATION : State

   EQUALS UNDEFINED IF ivReset = TRUE

   EQUALS PowerOff IF
      TABLE
         @T(..AltitudeStatus = Above) : T *;
         ..AltitudeStatus = Above     : * T;
         ..ASWOpModes = Inhibited     : F *;
         @F(..ASWOpModes = Inhibited) : * T;
        ivReset                       : F *;
      END TABLE

   EQUALS PowerOn IF
      TABLE
         @T(..AltitudeStatus = Below)           : T;
         ..ASWOpModes = Inhibited               : F;
         ivReset                                : F;
         PREV_STEP(..AltitudeStatus) = Unknown  : F;
      END TABLE
END STATE_VARIABLE
```

Figure 6. The definition of the *DOI_Intended* **state variable.**

Analysis: Figure 7 shows a major part of the NuSMV code for the ASW system automatically translated from the ASW specifications written in RSML^{-e}. Note that this is the code for one instance of the ASW—not the code we will create to represent the two ASW systems for deviation analysis.

Given this system model, we would like to check the tolerance of the system in terms of deviation of the measured altitude. Suppose one of the altimeters is not accurate and the measured altitude can be deviated by *-100 ft . . . 100 ft* from the actual value of the altitude. The main function of the ASW system is to signal the *On* command to the DOI when the aircraft descends below the threshold altitude. Since it is quite critical that the DOI is turned on in a timely manner, we would like the ASW to tolerate deviations in the altitude measures. In particular, we want to make sure that the DOI is never turned on 'too late'. This can be captured as the property *"The deviated system signals the DOI command* On *whenever the correct system signals the DOI command* On*"*. Note here that we are not concerned about the deviation leading to the DOI being turned on 'too early'; this is an acceptable performance degradation in the face of deviations, 'too late', however, is not acceptable.

To achieve this level of fault tolerance we will have to include more than one altimeter in the ASW system—a positive deviation with only one altimeter will always lead to the DOI being turned on too late. Therefore, we have included three redundant altimeters and use a voting scheme to determine if we are above or below the threshold. A voting scheme where we require all altimeters to be below the threshold before we turn the DOI on will, not surprisingly, be useless since a positive deviation in any altimeter will lead to the DOI being turned on too late. Therefore, we selected a voting scheme that will 'obviously' solve our problem—we will turn the DOI on as soon as *one* altimeter indicates we are below the threshold and we will turn the DOI off as soon as *all* altimeters indicate we are above the threshold plus hysteresis. With this voting scheme, the DOI will be turned on early of we have a negative deviation in one altimeter and there will be no change in when the DOI is turned on if we have a positive deviation. Also, the DOI may be turned off late if we have a negative deviation in one altimeter and there will be no change in when the DOI is turned off if we have a positive deviation (see Figure 8)—at least that is what we expected before applying our deviation analysis.

The NuSMV code is translated using the embedding scheme described in the previous section as shown in Figure 9. The main module defines input variables for the ASW system and the range of deviations for one of the altimeters. It embeds two synchronous sub-processes *ASW_Original* and *ASW_Deviated* that accept the values of input variables defined in the main module; correct values for the process *ASW_Original* and deviated values for the process *ASW_Deviated*. The definition for the sub-module *ASW* is identical to the original ASW system definition except for

```
MODULE main
DEFINE
 --- declare constants ---
AltitudeThreshold:= 2000;
Hysteresis:= 200;
DOIDelay:= 2;
AltBadTolerance:= 5;

VAR
 --- declare input variables ---
Altitude1 : 0..40000 ;
AltitudeQ1:{Good,Bad,Un_defined } ;
Altitude2 : 0..40000 ;
AltitudeQ2:{Good,Bad,Un_defined } ;
Altitude3 : 0..40000 ;
AltitudeQ3:{Good,Bad,Un_defined } ;
InhibitSignal: {Inhibit,NoInhibit};
ivReset : boolean;

--- declare state variables ---
DOICommand: {On,Off,Un_defined } ;
AltitudeStatus: {Unknown,Above,Below,AltitudeBad,Un_defined } ;
ASWOpModes: {OK,Inhibited,FailureDetected,Un_defined } ;
FaultDetectedVariable: {0, 1, Un_defined } ;
DOI_Intended: {PowerOff,PowerOn,Un_defined } ;

--- declare macro variables ---
m_BelowThreshold:BelowThreshold(AltitudeThreshold,Altitude1,AltitudeQ1,Altitude2,AltitudeQ2,Altitude3,AltitudeQ3);
m_AltitudeQualityOK:AltitudeQualityOK(AltitudeQ1,AltitudeQ2,AltitudeQ3);
m_AboveThresholdHyst:AboveThresholdHyst(AltitudeThreshold,Hysteresis,Altitude1,AltitudeQ1,Altitude2,AltitudeQ2,Altitude3,AltitudeQ3);

ASSIGN
init(InhibitSignal):= NoInhibit;
init(ivReset):= 0;
init(AltitudeStatus):=Un_defined;
init(DOICommand):=Un_defined;

--- state variable assignments ---
next(DOICommand):=
 case
      (((next(DOI_Intended)=PowerOn) & !((DOI_Intended=PowerOn))))  :  On;
      (((next(DOI_Intended)=PowerOff) & !((DOI_Intended=PowerOff)))) :  Off;
      1                                                               : DOICommand ;
 esac;

next(AltitudeStatus):=
 case
      (((AltitudeStatus= Un_defined ))) | (((next(ivReset))))                                     :  Unknown;
      ((next(m_BelowThreshold.result))&(next(m_AltitudeQualityOK.result))&!((next(ivReset))))      :   Below;
      ((next(m_AboveThresholdHyst.result))&(next(m_AltitudeQualityOK.result))&!((next(ivReset))))  :  Above;
      (!(next(m_AltitudeQualityOK.result))&!((next(ivReset))))                                     : AltitudeBad;
      1                                                                                            : AltitudeStatus ;
 esac;

next(ASWOpModes):=
 case
     (((next(ivReset))))                                                                                                     :   Un_defined ;
     (((next(InhibitSignal)=Inhibit)) & !((next(ivReset))))                                                                   :  Inhibited;
     (!((next(InhibitSignal)=Inhibit)) & ((next(AltitudeStatus)=AltitudeBad)) & !((next(ivReset))))                          :  FailureDetected;
     (!((ASWOpModes=FailureDetected)) & !((next(InhibitSignal)=Inhibit)) & !((next(AltitudeStatus)=AltitudeBad)) & !((next(ivReset))))   :  OK;
     1                                                                                                                        : ASWOpModes ;
 esac;

next(FaultDetectedVariable):=
 case
    (((next(ASWOpModes)=FailureDetected)))    :  1;
    (((next(ASWOpModes)=OK)))                 :  0;
    1                                          : FaultDetectedVariable ;
 esac;

next(DOI_Intended):=
 case
   ((((next(ivReset))=1)))                                                                                                  :   Un_defined ;
   ((next(AltitudeStatus)=Below) &  !((AltitudeStatus=Below)))
      &!((next(ASWOpModes)=Inhibited))&!((next(ivReset)))&!((AltitudeStatus=Unknown)))                                     :  PowerOn;
   (((next(AltitudeStatus)=Above))&( !((next(ASWOpModes)=Inhibited)) & (ASWOpModes=Inhibited)))|(((next(AltitudeStatus)=Above) &
      !((AltitudeStatus=Above)))&!((next(ASWOpModes)=Inhibited))&!((next(ivReset))))                                        :  PowerOff;
   1                                                                                                                        : DOI_Intended ;
 esac;

MODULE BelowThreshold(AltitudeThreshold,Altitude1,AltitudeQ1,Altitude2,,AltitudeQ2,Altitude3,AltitudeQ3)
VAR
 result : boolean;

ASSIGN
   init(result):=0 ;
   next(result):=    (((next(AltitudeQ3)=Good)) & ((next(Altitude3)<AltitudeThreshold))) | (((next(AltitudeQ2)=Good))&((next(Altitude2)<AltitudeThreshold)))
                     | (((next(AltitudeQ1)=Good)) & ((next(Altitude1)<AltitudeThreshold)));

MODULE AboveThresholdHyst(AltitudeThreshold,Hysteresis,Altitude1,AltitudeQ1,Altitude2,AltitudeQ2,Altitude3,AltitudeQ3)
VAR
 result : boolean;

ASSIGN
init(result):=0 ;
next(result):=  /* true if all of the altitude values are above the threshold hysteresis
                     false , other wise */
```

Figure 7. Fraction of NuSMV code for the ASW system

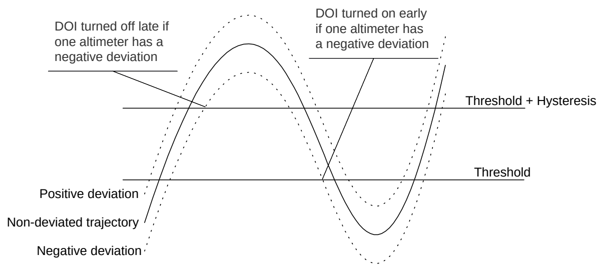
DOI turned off late if one altimeter has a negative deviation
DOI turned on early if one altimeter has a negative deviation
Threshold + Hysteresis
Threshold
Positive deviation
Non-deviated trajectory
Negative deviation

Figure 8. A negative deviation will turn on the DOI early. A positive deviation will have no effect.

the removal of the input variable declarations and the macro declarations; the macro declarations are referenced from both sub-processes and do not need to be declared twice.

```
MODULE main
DEFINE
  Altitude1_Deviated := Altitude1 + Deviation;

VAR
 --- declare input variables ---
  Altitude1 : 0..40000 ;
  AltitudeQ1:{Good,Bad,Un_defined } ;
  Altitude2 : 0..40000 ;
  AltitudeQ2:{Good,Bad,Un_defined } ;
  Altitude3 : 0..40000 ;
  AltitudeQ3:{Good,Bad,Un_defined } ;
  InhibitSignal: {Inhibit,NoInhibit};
  ivReset : boolean;

--- declare deviation limit
  Deviation: -100..100;
--- declare sub-systems
 ASW_Original : ASW(Altitude1, AltitudeQ1,Altitude2,AltitudeQ2,
                          Altitude3, AltitudeQ3,InhibitSignal,ivReset);
 ASW_Deviated : ASW(Altitude1_Deviated, AltitudeQ1,Altitude2,
                         AltitudeQ2, Altitude3,AltitudeQ3,InhibitSignal,ivReset);
SPEC
  AG(ASW_Original.DOICommand=On -> ASW_Deviated.DOICommand=On);

--- subsystem definition
MODULE ASW(Altitude1, AltitudeQ,Altitude2,AltitudeQ2,
                          Altitude3, AltitudeQ3,InhibitSignal,ivReset)
/* the code is the same as the code in the original ASW system except for the
   removal of the input variable declaration and the macro declaration */

/* the common macro declaration part */
MODULE BelowThreshold(..)
....
.....
```

Figure 9. NuSMV code for deviation analysis.

The property *"The deviated system signals the DOI command* On *whenever the correct system signals the DOI command* On*"* can be specified in CTL as

AG(ASW_Original.DOICommand=On $\rightarrow$
  ASW_Deviated.DOICommand=On)

Since the model includes several integer variables over a large domain, such as *Altitude1: 0..40000;*, model checking the ASW system is not feasible without using some abstraction. The change of the integer values in the ASW is not constrained, i.e., the altitude values are random input. Therefore, we can apply a simple *domain reduction abstraction* [1] to reduce the size of the domain without affecting the behavior of the system.

At a high level, the idea behind the simplest version of domain reduction abstraction is to partition the input domain based in the collection of numeric guarding conditions in the model. We then reduce the domain to a set of random representatives, one from each equivalence class. In the ASW mode we can identify six numeric guarding conditions.

$$\begin{array}{lcl}
Altitude1 & < & AltitudeThreshold \\
Altitude1 & > & AltitudeThreshold + Hysteresis \\
Altitude2 & < & AltitudeThreshold \\
Altitude2 & > & AltitudeThreshold + Hysteresis \\
Altitude3 & < & AltitudeThreshold \\
Altitude3 & > & AltitudeThreshold + Hysteresis
\end{array}$$

The constraints produce the following data equivalence classes.

$$\begin{array}{lcl}
a_{i1} & : & Altitude\#i < AltitudeThreshold \\
a_{i2} & : & Altitude\#i \geq AltitudeThreshold \wedge \\
 & & Altitude\#i \leq AltitudeThreshold + Hysteresis \\
a_{i3} & : & Altitude\#i > AltitudeThreshold + Hysteresis
\end{array}$$

where $i = 1..3$. After selecting a representative value from each equivalence class, the domain of each altitude variable is reduced to *Altitude#i :* $\{$*1999, 2001, 2201*$\}$. We proved in [1] that a system model with such a reduced domain bisimulates the original system model.

After applying the domain reduction abstraction, NuSMV easily checks the property and generates a counter example as shown in Figure 10. The variables with a d subscript in the lower half of the table are the variables in the deviated system—there is a *[-100..100]* deviation in *Altitude1*. The issue highlighted by the counter example is a startup problem caused by our definition of the initial system behavior.

A graphical view of the startup scenario can be seen in Figure 11. At system startup, the state variables *AltitudeStatus* and *DOICommand* are given the value *Undefined* since we do not know if we are above or below the threshold and, consequently, we do not know if the DOI should be on or off. In this initial version of the ASW, we do not assign a new value to the *DOICommand* until we cross one of the thresholds (either we drop below the threshold or we raise above the threshold plus hysteresis)—note that we turn the DOI on and off based on the *event* of crossing the thresholds, not based on the conditions of being above

Variable/Step	1	2	3	4
Altitude1	*Undefined*	.	2201	1999
Altitude2	*Undefined*	.	2201	1999
Altitude3	*Undefined*	.	2201	1999
AltitudeStatus	*Undefined*	*Unknown*	*Above*	*Below*
DOICommand	*Undefined*	*Undefined*	*Off*	*On*
Altitude1_d	*Undefined*	.	2191	1999
Altitude2_d	*Undefined*	.	2201	1999
Altitude3_d	*Undefined*	.	2201	1999
AltitudeStatus_d	*Undefined*	*Unknown*	*Unknown*	*Below*
DOICommand_d	*Undefined*	*Undefined*	*Undefined*	*Undefined*

Figure 10. A counter example trace

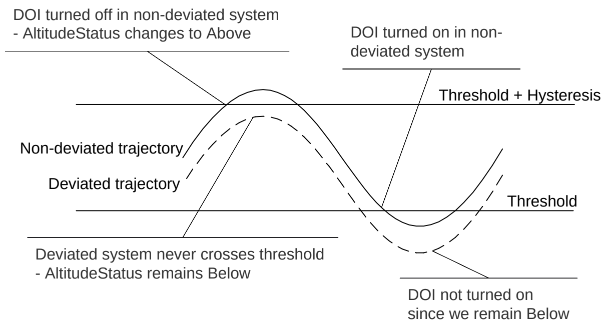
DOI turned off in non-deviated system
- AltitudeStatus changes to Above
DOI turned on in non-deviated system
Threshold + Hysteresis
Non-deviated trajectory
Deviated trajectory
Threshold
Deviated system never crosses threshold
- AltitudeStatus remains Below
DOI not turned on
since we remain Below

Figure 11. The startup scenario problem.

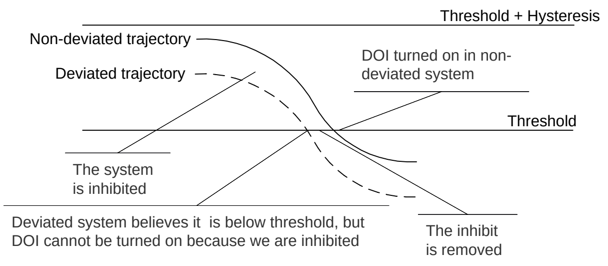
Threshold + Hysteresis
Non-deviated trajectory
Deviated trajectory
DOI turned on in non-deviated system
Threshold
The system
is inhibited
Deviated system believes it is below threshold, but
DOI cannot be turned on because we are inhibited
The inhibit
is removed

Figure 12. The inhibit scenario problem.

Variable/Step	1	2	3	4
Altitude1	*Undefined*	.	2201	1999
Altitude2	*Undefined*	.	2201	1999
Altitude3	*Undefined*	.	2201	1999
AltitudeStatus	*Undefined*	*Unknown*	*Above*	*Below*
DOICommand	*Undefined*	*Undefined*	*Off*	*On*
Altitude1_d	*Undefined*	.	2191	1999
Altitude2_d	*Undefined*	.	2201	1999
Altitude3_d	*Undefined*	.	2201	1999
AltitudeStatus_d	*Undefined*	*Unknown*	*Unknown*	*Below*
DOICommand_d	*Undefined*	*Undefined*	*Undefined*	*On*

Figure 13. Corresponding trace after correction

or below. Therefore, the value of the *DOICommand* does not change to *Off* until the *AltitudeStatus* becomes *Above*—the *DOICommand* will remain *Undefined* until this event happens. The counter example shows that if we have a negative deviation, the original system raises above the threshold plus hysteresis, thus setting the *AltitudeStatus* to *Above* and the *DOICommand* to *Off*. The deviated system, on the other hand, is still considered to be below the threshold because of the negative deviation so no action is taken. When the aircraft now descends below the threshold, the original system's *AltitudeStatus* will change from *Above* to *Below*—an event that will cause the DOI to be turned on. Since the deviated system never changed *AltitudeStatus* to *Above*, the event of changing from *Above* to *Below* will never take place and, consequently, the DOI will not be turned on. We have discovered how a critical function can be effected by a deviation in one altimeter despite our conservative voting mechanism.

After analyzing the counter example, one would expect that the system would tolerate the deviation if we changed the startup behavior of the system—we will now allow the DOI to be turned on immediately at startup if we are below the threshold and off if we are above threshold hysteresis no matter what the previous value of *AltitudeStatus* was; at startup we will no longer wait for the event of crossing the thresholds to occur. With this modification in startup behavior the problem is solved, the trace in Figure 10 would become the trace shown in Figure 13. The problem, however, was not that simple; the model checker quickly found another counter example trace related to the inhibit signal that prevents the system from issuing output commands. The counter example in Figure 14 shows this case (a graphical illustration is available in Figure 12); if we have a negative deviation in one altimeter, the value of *AltitudeStatus* of the deviated system becomes *Below* in the second state because of the deviation (the deviated variable is less than the threshold) but the ASW cannot set the value of *DOICommand* to *On* since it is inhibited. The original system stays above the threshold in this state. In the next state, the aircraft descends below the threshold and the inhibit is removed. The original system can set *DOICommand* to *On* since it is not inhibited and the event *Above* to *Below* occurred, but the deviated system still cannot set *DOICommand* to *On* since in this system the event happened while it was inhibited.

When this problems is corrected, a similar issue is raised with an ASW reset function that is designed to bring the system back to its initial state. Although our ASW can be corrected so that it does tolerate deviations in one altimeter, the example serves to demonstrate how a fault tolerance mechanism that will 'obviously correct the problem' exhibits undesirable behavior under various non-obvious circumstances. In our limited experience with deviation analysis, the problems exposed seem to be related to startup

Variable/Step	1	2	3	4
Altitude1	*Undefined*	.	2001	1999
Altitude2	*Undefined*	.	2001	1999
Altitude3	*Undefined*	.	2001	1999
AltitudeStatus	*Undefined*	*Unknown*	*Unknown*	*Below*
Inhibit	*Undefined*	.	*Inhibited*	*NotInhibited*
DOICommand	*Undefined*	*Undefined*	*Undefined*	*On*
Altitude1_d	*Undefined*	.	1964	2026
Altitude2_d	*Undefined*	.	2001	1999
Altitude3_d	*Undefined*	.	2001	1999
AltitudeStatus_d	*Undefined*	*Unknown*	*Below*	*Below*
Inhibit_d	*Undefined*	.	*Inhibited*	*NotInhibited*
DOICommand_d	*Undefined*	*Undefined*	*Undefined*	*Undefined*

Figure 14. Counter example trace after correction

behaviors, temporary shutdowns and inhibits, and system reset behaviors—well known problem areas in critical systems [4].

6. Discussion

In this paper we reported on an effort to perform deviation analysis using standard model checkers. Our work is an alternative to other approaches based on a symbolic execution of the system models and promises to provide a more accurate analysis than what was previously possible. In our, admittedly very limited, experience, deviation analysis through model checking works well and has helped us identify problems in smaller examples. More work is necessary before the feasibility of the approach on larger problems can be determined. The future challenges mainly fall in two categories: comparative evaluation and conquering the state space explosion problem.

Here, we showed that deviation analysis through model checking can be effective in pointing out subtle problems in a system model. We did not, however, make any claims as to the relative effectiveness of tackling real world safety analysis problems with our approach compared to other proposed techniques. We believe the exploratory nature of the original devaition/perturbation analysis will nicely complement the more verification-oriented nature of deviation analysis through model checking. The interaction of the techniques, and a possible incorporation of theorem proving and model checking techniques in our perturbation analysis are issues worth further study.

The size of the representation of the state space and the next state relation are the limiting factor when model checking larger systems. Since we are in essence simultaneously analyzing two copies of a system (correct and deviated), we have many more variables to contend with. We hope, however, that existing abstraction techniques in conjunction with techniques currently under development at the University of Minnesota will help address this problem so that deviation analysis through model checking will become usable on realistic systems.

References

[1] Y. Choi, S. Rayadurgam, and M. Heimdahl. Automatic abstraction for model checking software systems with interrelated numeric constraints. In *Proceedings of the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE-9)*, pages 164–174, September 2001.

[2] CISHEC. *A Guide to Hazard and Operability Studies*. The Chemical Industry Safety and Health Council of the Chemical Industries Association Ltd., 1977.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[4] M. S. Jaffe, N. G. Leveson, M. P. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[5] T. Kletz. *Hazop and Hazan: Identifying and Assessing Process Industry Standards*. Institution of Chemical Engineers, 1992.

[6] J. McDermid and D. J. Pumfrey. A development of hazard analysis to aid software design. In *COMPASS '94: Proceedings of the Ninth Annual Conference on Computer Assurance*, pages 17–25. IEEE/NIST, June 1994.

[7] S. P. Miller and A. C. Tribble. Extending the four-variable model to bridge the system-software gap. In *Proceedings of the Twentith IEEE/AIAA Digital Avionics Systems Conference (DASC'01)*, October 2001.

[8] NuSMV: A New Symbolic Model Checking. Available at http://nusmv.irst.itc.it/.

[9] J. Reese and N. Leveson. Software deviation analysis. In *International Conference on Software Engineering*, May 1997.

[10] J. Reese and N. Leveson. Software deviation analysis: A "safeware" technique. In *AIChe 31st Annual Loss Prevention Symposium*, March 1997.

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

[12] J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[13] M. W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.