
Automated Software Engineering, 12, 321–347, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Deviation Analysis: A New Use of Model Checking

MATS P.E. HEIMDAHL
YUNJA CHOI
MICHAEL W. WHALEN
Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., 4-192, Minneapolis, MN 55455, USA

Abstract. Inaccuracies, or deviations, in the measurements of monitored variables in a control system are facts of life that control software must accommodate. *Deviation analysis* can be used to determine how a software specification will behave in the face of such deviations. Deviation analysis is intended to answer questions such as "*What is the effect on output O if input I is off by 0 to 100?*". This property is best checked with some form of symbolic execution approach. In this report we wish to propose a new approach to deviation analysis using model checking techniques. The key observation that allows us to use model checkers is that the property can be restated as "*Will there be an effect on output O if input I is off by 0 to 100?*"—this restatement of the property changes the analysis from an exploratory analysis to a verification task suitable for model checking.

Keywords: deviation analysis, model checking, robustness

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, faulty switches, 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 (Reese and Leveson, 1997a, b; Reese, 1996) 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 (Clarke et al., 1999). 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 previously suggested approaches to deviation analysis and supports an alternate verification style. We have defined the mechanism for performing this analysis, implemented a prototype tool, and evaluated the approach on several case examples.

2. Background

Reese and Leveson introduced the notion of *software deviation analysis* in 1996 (Reese, 1996). The method is based on Hazard and Operability (HAZOP) analysis (CISHEC, 1977; Kletz, 1992), 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.

2.1. Hazard and Operability analysis (HAZOP)

Developed for the British chemical industry in the 1950's, HAZard and OPerability analysis (HAZOP) (CISHEC, 1977; Kletz, 1992) 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., "manifold pressure," yielding a question: "*What is the potential effect of the pressure in the manifold 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 (McDermid and Pumfrey, 1994), 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 pipe diagrams) 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.*

2.2. Software Deviation Analysis

Software Deviation Analysis (Reese and Leveson, 1997a; Reese, 1996) 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 black-box 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 Reese (1996), 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 reading on altimeter 1 is 'Very-High-Positive', what will be the effect on the pitch command?" Deviation analysis, as well as the perturbation analysis discussed below, are based on abstracting away the details of the computations and are thus closely related to abstract interpretation (Cousot and Cousot, 1977).

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, in collaboration with Reese and Leveson we developed a related approach based on interval calculus as opposed to qualitative mathematics.

2.3. Perturbation analysis

Perturbation analysis is an adaptation and simplification of Reese's deviation procedure for the RSML^{-e} language. RSML^{-e} (Thompson et al., 1999; Whalen, 2000) 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 to decide to what extent x and y are perturbed. The user provides a range of correct values and also a range of perturbation. In the analysis, we prompt the user for all values that cannot be directly computed in this step: the values of input variables, the time input variables were assigned, and values of state variables in the previous step (if this information is relevant). For our example, a user could define x and y as follows:

Variable	Correct range	Perturbation	Perturbed 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 from the analysis a set of ⟨*condition*, *correct*, *perturbed*⟩ tuples, where if *condition* holds, then *correct* describes the possible correct range of values and *perturbed* describes the maximum perturbation possible. First, we can determine the normal values of z by using interval arithmetic:

– If b, then the normal range of z is $x \times 2 = [0..10] \times 2 = [0..20]$
– If $\neg b$ and $\neg c$, then the normal range of z is $y = [5..60]$
– If $\neg b$ and c, then the normal range of z is $x \times 2 - y = [0..20] - [5..60] = [-60..15]$

Given perturbations in x and y,we can also determine the maximum perturbations of z by the same process.

– If b, then the perturbed range of z is $x \times 2 = [-1..13] \times 2 = [-2..26]$
– If $\neg b$ and $\neg c$, then the perturbed range of z is $y = [15..80]$
– If $\neg b$ and c, then the perturbed range of z is $x \times 2 - y = [-2..26] - [15 - 80] = [-82..11]$

To summarize, the deviations in z given deviations of x and y are given by the tuples below:

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]

The idea behind this analysis is appealing since it helps answer many questions during safety analysis that can be very difficult to address without tool support. Unfortunately, as discussed below, there were several issues that led us to abandon perturbation analysis before we completed a stable prototype or any publications.

2.4. Issues with existing approaches

These analyses, while useful, have issues that, in our opinion, 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.

Second, 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.

Third, the reports from our prototype tool were excessively large. Our implementation did not check whether or not the *condition* of each pair is satisfiable. Given decision procedures, it would be possible to reduce the number of cases reported by the symbolic execution by removing unsatisfiable conditions. In our experience, however, decision procedures alone will not remove enough spurious reports—the size of the output may still be overwhelming in even a small system. We could further reduce the number of unsatisfiable cases by checking whether certain conditions were possible once the intervals for input and state variables are given. There are, however, some problems with extending the standard relational operators over intervals that make it difficult to create accurate decision procedures. For example, defining various operators such as the notion of equivalence of intervals poses a problem—shall two intervals be equivalent if they have the same upper and lower bound, or is it enough that they overlap? Either choice is defensible, but both may yield spurious reports and may falsify predicates that are satisfiable in the original system.

Finally, the perturbation analysis was originally defined is a 'one-step' analysis; it does not record how a series of perturbations affect the specification over time. Unfortunately,

many of the properties in which one might be interested (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.

2.5. Recent developments

Recently, Ait-Ameur et al., developed an approach similar to perturbation analysis based on abstract interpretation of LUSTRE programs (Ait-Ameur et al., 2003). In this work, they focus on the effect of deviations in real variables that are inputs to control laws implemented in the LUSTRE language (Halbwachs et al., 1991). They define an abstract interpretation (Cousot and Cousot, 1977) of the real domain in LUSTRE programs—using interval calculus—that allows an analyst to compute approximations of the deviations in output variables caused by deviations in inputs. The analysis has been implemented in a prototype Java tool. This tool has been applied to the various LUSTRE libraries available at Airbus Industries. Their approach is exciting, but suffers from problems similar to the ones we faced with perturbation analysis (see above). It is likely that this approach will scale better than an approach based on model checking (and works with real-valued variables), but at the cost of less precision. Continued refinement of the work presented in Ait-Ameur et al. (2003), possibly coupled with constraint solving and decision procedures, may yield a complementary approach for analyzing the robustness of software systems in the face of deviated inputs.

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

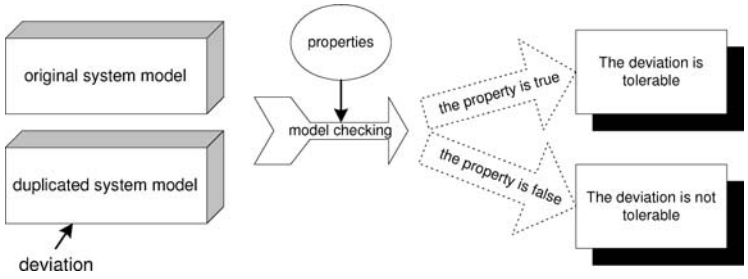

Figure 1. General approach of deviation analysis using model checking.

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 impose 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.

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.

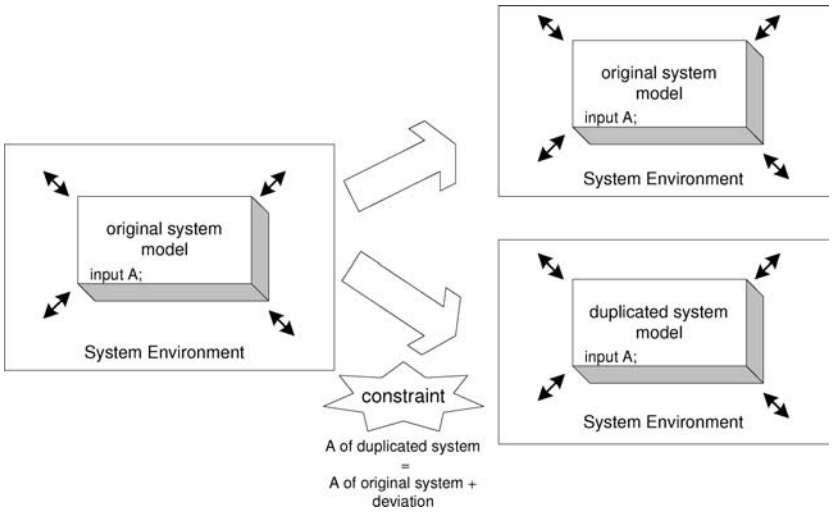

Figure 2. Modeling Scheme 1—simple duplication.

4.1. 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 (NuSMV,) as follows (variables with a *d* subscript represents the variables in the deviated system):

$$\text{INVAR } Y_d = Y$$
$$\text{INVAR } X_d = X + \text{deviation}$$

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.

4.2. 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

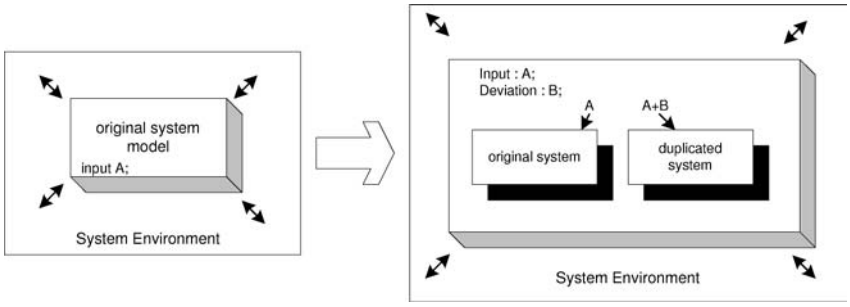

Figure 3. Modeling Scheme 2—embedding.

scheme \ usage	# 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 modeling schemes.

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 our sample system, the Altitude Switch, (after applying abstraction on numeric variables as described in the next section). Since the embedding scheme is significantly more efficient, we have chosen to pursue this approach for our prototype tool. Our tool uses RSML^{-e} as the source language and produces output to NuSMV (NuSMV,); nevertheless, our general approach is not limited to any particular model checking tools.

5. Examples

To illustrate our approach to deviation analysis, we will use two examples from the avionics domain—the Altitude Switch and a Flight Guidance System. Both systems we are using

as our examples have their origin with Rockwell-Collins' Advanced Technology Center in Cedar Rapids, Iowa. Using the altitude switch example, we will illustrate how deviation analysis through model checking can be used to detect subtle problems with straight forward and seemingly correct approaches to tolerating altitude measurement deviations. The flight guidance system will be used to show how deviation analysis can be used to analyze a complex system for mode confusion properties.

5.1. The altitude switch

The Altitude Switch (ASW) is a hypothetical device based on a real system 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 Miller and Tribble, 2001 and Thompson et al., 1999). The robustness to deviations in the altitude measures is the subject of interest in the following two subsections.

The version of the ASW used in this paper receives altitude information from some number of radio altimeters (figure 5). The functionality 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 6. Using the textual RSML^{-e} syntax, we view the *AltitudeStatus* to be *Unknown* at system startup or after a reset. In addition, we define *AltitudeStatus* to be a child variable of the state *PowerStatus.On* (indicated by the parent field), that is, *AltitudeStatus* is only relevant when the ASW power is on. *AltitudeStatus* is assigned the value in an EQUALS clause when the guard condition in that clause is true. Should more than one guard be satisfied, the first EQUALS clause is chosen. 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

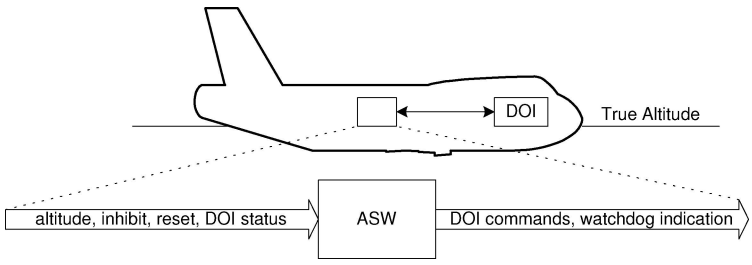

Figure 5. The ASW system in its environment.

```
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


MACRO BelowThreshold() :
   TABLE
      AltitudeQ1 = Good             : T * *;
      Altitude1 = UNDEFINED         : F * *;
      Altitude1 < AltitudeThreshold : T * *;
      AltitudeQ2 = Good             : * T *;
      Altitude2 = UNDEFINED         : * F *;
      Altitude2 < AltitudeThreshold : * T *;
      AltitudeQ3 = Good             : * * T;
      Altitude3 = UNDEFINED         : * * F;
      Altitude3 < AltitudeThreshold : * * T;
   END TABLE
END MACRO
```

Figure 6. The definition of the *AltitudeStatus* state variable and the *BelowThreshold()* macro.

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 sample macro in figure 6 shows the voting scheme used in this example, as soon as one of our three altimeters report an altitude below the threshold we consider ourselves below). The interested reader can find a complete description and formal semantics of the RSML^{-e} language in Whalen (2005)

The ASW command to the Device of Interest (DOI) is defined as in figure 7. At startup and after a reset, we do not know what 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 inhibit is removed. We turn on the DOI if the system is operational (i.e., it is not inhibited or reset) and the altitude changed from above to below (condition rows 1 and 4).

5.1.1. The effect of altitude deviations on the DOI. Figure 8 shows a major part of the NuSMV code for the ASW system automatically translated from the ASW specifications

```
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 7. The definition of the *DOI_Intended* state variable.

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 if 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 9)—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 10. 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 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.

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

P1. AG(ASW_Original.DOICommand=On $\rightarrow$
ASW_Deviated.DOICommand=On)

```
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 8. Fraction of NuSMV code for the ASW system.

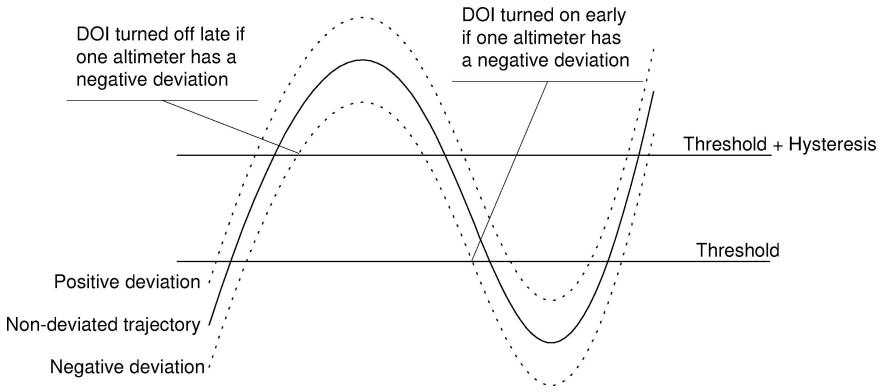

Figure 9. A negative deviation will turn on the DOI early. A positive deviation will have no effect.

```
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 10. NuSMV code for deviation analysis.

Since the model includes several integer variables over large domains, 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* (Choi et al., 2002; Choi, 2003) 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.

$$
\begin{aligned}
&\mathit{Altitude}1 < \mathit{AltitudeThreshold}\\
&\mathit{Altitude}1 > \mathit{AltitudeThreshold} + \mathit{Hysteresis}\\
&\mathit{Altitude}2 < \mathit{AltitudeThreshold}\\
&\mathit{Altitude}2 > \mathit{AltitudeThreshold} + \mathit{Hysteresis}\\
&\mathit{Altitude}3 < \mathit{AltitudeThreshold}\\
&\mathit{Altitude}3 > \mathit{AltitudeThreshold} + \mathit{Hysteresis}
\end{aligned}
$$

The constraints produce the following data equivalence classes.

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

where $i = 1..3$. After selecting a representative value from each equivalence class, the domain of each altitude variable is reduced to $\mathit{Altitude\#i} : \{\mathit{1999, 2001, 2201}\}$. We proved in Choi et al. (2001) that a system model with such a reduced domain bi-simulates the original system model.

After applying the domain reduction abstraction and extending the domain of the altitude variables to accommodate the specified deviation, NuSMV easily checks the property and generates a counter example as shown in figure 11. 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 12. 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 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

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 11. A counter example trace.

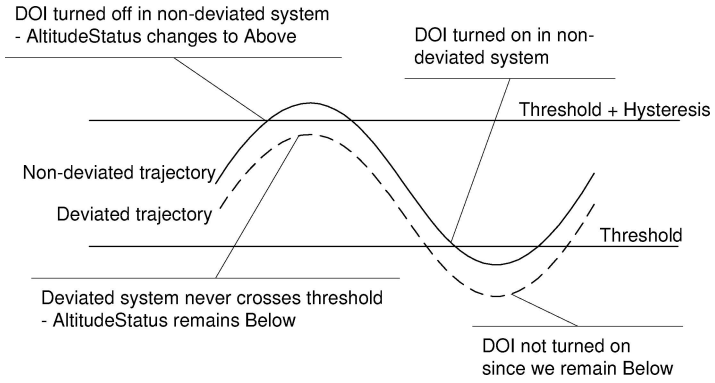

Figure 12. The startup scenario problem.

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

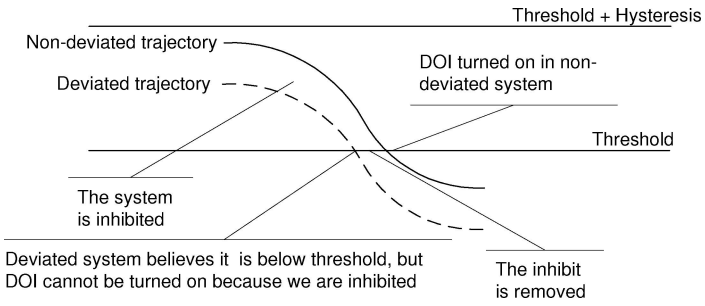

Figure 13. The inhibit scenario problem.

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 14. Corresponding trace after correction.

are above the threshold plus 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 11 would become the trace shown in figure 14. 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 15 shows this case (a graphical illustration is available in figure 13); 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.

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 15. Counter example trace after correction.

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 behaviors, temporary shutdowns and inhibits, and system reset behaviors—well known problem areas in critical systems (Jaffe et al., 1991).

5.1.2. The effect of altitude deviations on altitude rate. Our approach to deviation analysis can also be used for quantitative analysis. For example, assume that the ASW is computing the altitude rate in addition to its duties related to the DOI. The altitude rate is simply the difference between two altitude readings divided by the time between the readings. This is an admittedly very simplistic example that nevertheless suffices to illustrate the main point. Further assume we want to answer the question "*If one altimeter has a measurement error of 0..100 ft, what will be the deviation of the computed altitude rate?*". We can change the question into a verification task by asking "*If one altimeter has a measurement error of 0..100 ft, the deviation in the altitude rate will be less than y ft?*". The main difference between the questions is that in the second question we first need to provide an acceptable value for the deviation of the altitude rate and then check the property using model checking. If the property does not hold (the deviation is larger than y) we know that the deviation in the input leads to an unacceptable deviation in the output. If we want to know how large the deviation in the output may be, we need to adjust the value of y upward and check the property again. This process will be repeated until we come up with the right value. This estimation process is admittedly inefficient and if we are interested in computing exact (or estimated) values of deviations rather than determining if a specific deviation is possible,

methods based on some form of symbolic execution would be preferable, for example, the approach based on abstract interpretation discussed in Ait-Ameur et al. (2003).

To demonstrate quantitative analysis, we modify the ASW system altitude computation scheme; the altitude value *Alt* is now defined as

$$Alt = \begin{cases} \dfrac{altitude1 + altitude2 + altitude3}{3} & \text{if all altimeters are OK} \\ \text{Undefined, otherwise} \end{cases}$$

The altitude rate is computed as

$$AltRate = Alt - PREV\_STEP(Alt)$$

since the altitude is sampled once per second. Again, we assume that the deviated system has a problem with Altimeter1; the altitude reports from altimeter1 are deviated by 0..100 from the true altitude.

We can now formulate our question "*If Altimeter1 has a measurement error of 0..100 ft, the deviation in the altitude rate will be less than 35 ft?*" as a CTL property

P2. AG(ASW_Deviated.AltRate − ASW_Original.AltRate < 35)

In order to verify the property, we again apply domain reduction abstraction to reduce the domain of the input variables as we discussed in the previous section. Note that we need to take the numeric conditions *ASW_Deviated.AltRate* − *ASW_Original.AltRate* < 35 into account from the property specification as well as the numeric conditions from the system model itself.

After applying the abstraction, property P2 quickly turns out false using NuSMV; if one of the altimeters is not functioning correctly (it signals that the altitude is bad), the averaged altitude takes on the value *Undefined*, and, consequently, *AltRate* becomes *Undefined* and the model checker assumes the value of *AltRate* is the same as the previous one which can be any value. In our property, we did not specify that the altitude rate used in the computations must actually be defined. Therefore, we adjusted the property as follows to require that all altimeters work correctly. This property proves to be true with our very simplistic scheme of averaging altitude reports.

P2.1. AG(ASW_Original.AltRate_Undefined = FALSE ⟶
ASW_Deviated.AltRate − ASW_Original.AltRate < 35)

In actuality, in the ASW we do not declare a failure until we have lost altitude reports from two or more altimeters—the ASW continues functioning normally with one failed altimeter. To capture this, we modify the logic so that the computation of altitude as well

Variable/Step	1	2	3
Altitude1	*Undefined*	2200	2201
Altitude2	*Undefined*	2200	2201
Altitude3	*Undefined*	2200	2201
AltitudeQ1	*Undefined*	*Good*	*Good*
AltitudeQ2	*Undefined*	*Good*	*Good*
AltitudeQ3	*Undefined*	*Good*	*Bad*
Alt	*Undefined*	2200	2201
AltRate	*Undefined*	*Undefined*	1
Altitude1_d	*Undefined*	2200	2301
Altitude2_d	*Undefined*	2200	2201
Altitude3_d	*Undefined*	2200	2201
Alt_d	*Undefined*	2200	2251
AltRate_d	*Undefined*	*Undefined*	51

Figure 16. A counter example for property P2.1.

as altitude rate tolerate failure of one altimeter.

$$Alt = \begin{cases} \dfrac{Altitude1 + Altitude2 + Altitude3}{3} & \textbf{if } AltitudeQ1 = AltitudeQ2 = \\ & \quad AltitudeQ3 = Good, \\ \dfrac{Altitude1 + Altitude2}{2} & \textbf{if } AltitudeQ1 = AltitudeQ2 = Good \ \& \\ & \quad AltitudeQ3 = Bad\ , \\ \dfrac{Altitude1 + Altitude3}{2} & \textbf{if } AltitudeQ1 = AltitudeQ3 = Good \ \& \\ & \quad AltitudeQ2 = Bad\ , \\ \dfrac{Altitude2 + Altitude3}{2} & \textbf{if } AltitudeQ2 = AltitudeQ3 = Good \ \& \\ & \quad AltitudeQ1 = Bad \end{cases}$$

NuSMV generates a counter example for property P2.1 after the modification of the altitude computation as shown in figure 16. If one of altimeters 2 or 3 fails, we continue computing an averaged altitude but we only average two altitude reports. Thus, a larger portion of the measurement error in Altimeter1 propagates to the altitude rate computation. After reviewing the counter example and the computation logic of *AltRate* we had to lower our expectations and accept a much larger possible deviation in the computed altitude rate than we previously planned for. This property is captured below as P2.2 and can be easily verified using NuSMV.

P2.2. AG(ASW_Original.AltRate_Undefined = FALSE $\longrightarrow$
ASW_Deviated.AltRate $-$ ASW_Original.AltRate < 55)

5.2. *Mode confusion analysis using a variation of deviation analysis*

Mode confusion refers to the situation where the operator of a system, in our case a pilot, gets confused about the status of the automation and starts interacting with it inappropriately (Leveson et al., 1997; Leveson and Palmer, 1997). In this section we will outline how a variation of deviation analysis can be used for analysis of certain mode confusion properties in a Flight Guidance System.

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS). It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generate pitch and roll guidance commands to minimize the difference between the measured and desired state.[3] The FGS can be broken down to mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands.

Figure 17 illustrates a graphical view of a FGS in the NIMBUS environment. The primary modes of interest in the FGS are the horizontal and vertical modes. The horizontal modes control the behavior of the aircraft about the longitudinal, or roll, axis, while the vertical

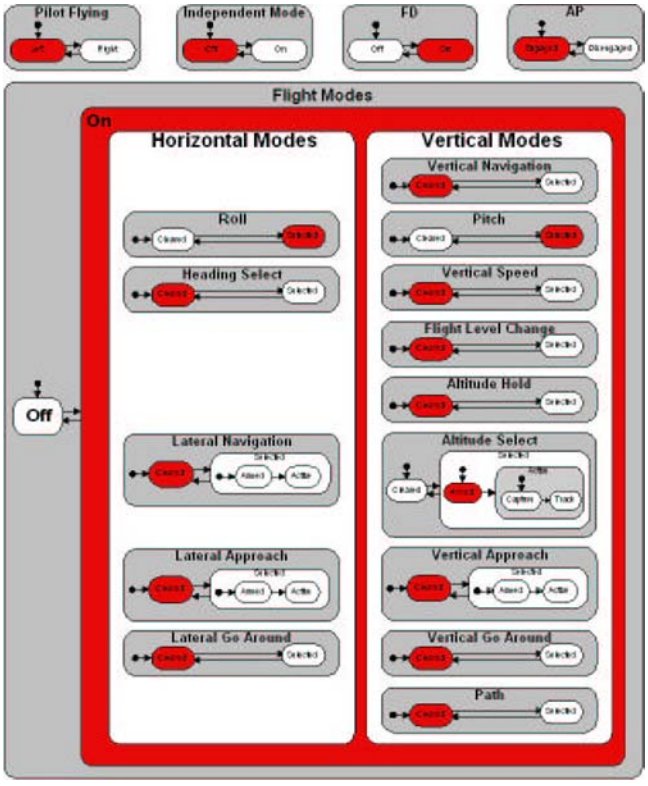

Figure 17. Flight guidance system.

modes control the behavior of the aircraft about the vertical, or pitch, axis. In addition, there are a number of auxiliary modes, such as half-bank mode, that control other aspects of the aircraft's behavior. Since the modes determine how the aircraft will be controlled, mode confusion problem becomes a critical issue—it is imperative that the pilot is provided adequate guidance as to which mode is actively controlling the aircraft.

Various research efforts have identified collections of conditions that might lead to mode confusion (Leveson et al., 1997; Leveson and Palmer, 1997; Miller and Potts, 1999). For example, it is well known that *hidden modes* are a serious problem—a mode is hidden if the system retains state information that may affect future mode changes, but this information is not annunciated to the pilot. This can be informally stated as the requirement below.

R3. *If the mode annunciations A are provided to the pilot and mode selection input I is given, the new mode annunciations A' will always be given.*

Note here that we have not really stated what the mode annunciations shall be or how they shall change. All we have stated is that *every time* the annunciations are A and we provide a mode switch input I, they will change to A'—the mode changes are consistent. This property cannot be checked using regular model checking techniques since we are interested in showing consistency of sets of variables over time, but we do not know the value of the variables. This property can be trivially formalized, however, if we can quantify over states.

P3. $\forall s_1, s_2 : (modeAnnunc(prev(s_1)) = modeAnnunc(prev(s_2))) \wedge (m\_inputs(s_1) = m\_inputs(s_2)) \Rightarrow modeAnnunc(s_1) = modeAnnunc(s_2)$

Nevertheless, we can check similar types of properties using a variation of the deviation analysis technique we have presented in this paper. For the purpose of mode confusion analysis, we introduce two identical models of the system and tie the mode annunciations in the previous state and the current mode switch inputs together to investigate if the current annunciations are consistent. Note that we are not introducing any artificial deviation in the model as we did for the general deviation analysis discussed earlier; we are not interested in investigating the response to deviations, instead we are interested in seeing if there is a potential for any unexpected mode changes to identical mode switch inputs. Therefore, as illustrated in figure 18, we treat the two system models as two independent systems, without introducing any extra constraints for the input variables as we did for deviation analysis.

To check a property similar to property P3, we must first identify the mode-annunciations and mode switch inputs in the system model. In the version of the FGS we used as an example, we identified the following variables:

Mode annunciations	HDG_Lamp, VS_Lamp, ALT_Lamp, AP_Lamp
Mode inputs	AP_Engage_Switch, AP_Disconnect_Switch, HDG_Switch, VS_Pitch_Wheel_In_Motion, VS_Switch, ALT_Switch, Transfer_Switch

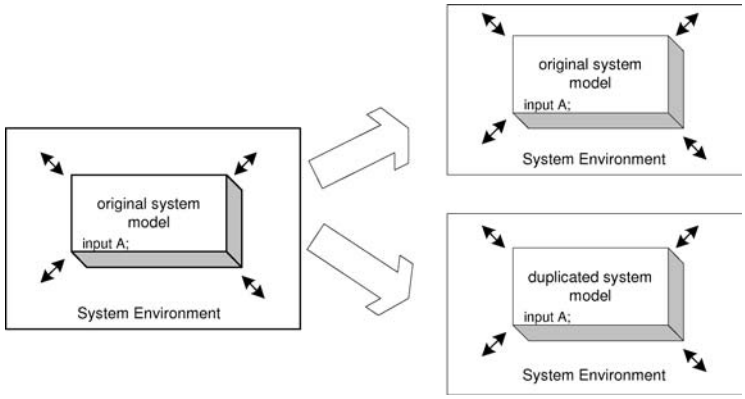

Figure 18. Two independent models for mode confusion analysis.

With a model that has two FGS system, FGS1 and FGS2 running synchronously, we can state a relaxed version of property P3 as property P4 below.

P4 AG(m_was_same_annunciations.result & m_same_inputs.result ⇒ m_same_annunciations .result)

where *m_was_same_annunciations.result*, *m_same_inputs.result*, and *m_same_annunciations. result* represent macros that compare values of mode annunciations and mode switch inputs of the two synchronously running FGS models. These macros are specified in NuSMV using its MODULE construct as shown in figure 19. Note here that P4 is a relaxation of P3. In P3 we are interested in if any two *arbitrary* states with the same mode annunciation will provide the same new mode annunciation after receiving identical mode switching commands. In the relaxed property P4, we are interested in if there there is a possibility of getting two different mode annunciations given identical sequences of mode switching commands—a situation that could occur if there is nondeterminism in the system or if other inputs affect the mode logic. We have not yet been able to investigate properties such as P3 in our deviation analysis framework. The version of the FGS we used in our case study was developed independently by Rockwell Collins Inc. using RSML^{-e} and has been extensively reviewed for mode confusion in a related project—thus, the deviation analysis proved the property above true.

6. Discussion

In this paper we reported on an effort to perform deviation analysis using standard model checkers. Our work is a complement 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 quite limited, experience, deviation analysis through

```
Same_annunciations
MODULE same_annunciations (HDG_Lamp1, VS_Lamp1, ALT_Lamp1, AP_Lamp1,
                           HDG_Lamp2, VS_Lamp2, ALT_Lamp2, AP_Lamp2)
VAR
result : boolean;

ASSIGN
init(result):= 0;
next(result):= (next(HDG_Lamp1)=next(HDG_Lamp2)) &
                      (next(VS_Lamp1)= next(VS_Lamp2)) &
                      (next(ALT_Lamp1) = next(ALT_Lamp2)) &
                      (next(AP_Lamp1) = next(AP_Lamp2));
```

```
Was_Same_annunciations
MODULE was_same_annunciations (HDG_Lamp1, VS_Lamp1, ALT_Lamp1,
                               AP_Lamp1,  HDG_Lamp2, VS_Lamp2,
                               ALT_Lamp2, AP_Lamp2)
VAR
result : boolean;

ASSIGN
init(result):= 0;
next(result):= (HDG_Lamp1=HDG_Lamp2) & (VS_Lamp1= VS_Lamp2) &
                      (ALT_Lamp1 = ALT_Lamp2) & (AP_Lamp1 = AP_Lamp2);
```

```
Same_Inputs
MODULE Same_inputs (AP_Engage_Switch1, AP_Disconnect_Switch1,
                    HDG_Switch1,
                    VS_Pitch_Wheel_In_Motion1, VS_Switch1, ALT_Switch1,
                    Transfer_Switch1, AP_Engage_Switch2,
                     AP_Disconnect_Switch2,
                    HDG_Switch2, VS_Pitch_Wheel_In_Motion2, VS_Switch2,
                    ALT_Switch2, Transfer_Switch2)
VAR
 result : boolean;

ASSIGN
 init(result) :=0;
 next(result):= (next(AP_Engage_Switch1)= next(AP_Engage_S  witch2) ) &
 (next(AP_Disconnect_Switch1)=next(AP_Disconnect_Switch2)) &
 (next(HDG_Switch1) = next(HDG_Switch2)) &
 (next(VS_Pitch_Wheel_In_Motion1) = next(VS_Pitch_Wheel_In_Motion2))  &
 ( next(VS_Switch1)= next(ALT_Switc  h1)) &
  (next(Transfer_Switch1) = next(Transfer_Switch2));
```

Figure 19. Macros for the mode confusion analysis.

model checking works well and has helped us identify problems in small to medium sized 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 deviation/perturbation analysis of Reese and Leveson (1997a, 1997b) and Reese (1996) and the abstract interpretation approach of Ait-Ameur et al. (2003) will nicely complement

the more verification-oriented nature of deviation analysis through model checking. The interaction of the techniques, and a possible incorporation of constraint solving and decision procedures in the exploratory techniques 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, and thus, our approach may not scale up to large systems. Given our experience with model checking realistic systems expressed in RSML^{-e} (Choi and Heimdahl, 2002), we believe the approach will scale as well as model checking in general. By adopting existing model checking abstraction techniques, such as iterative refinement abstraction (Clarke et al., 2000) and domain reduction abstraction (Choi et al., 2002), or other analysis approaches, such as bounded model checking (Biere et al., 1999), we hope to extend the scalability of our approach to larger systems.

Acknowledgments

This work has been partially supported by NASA grant NAG-1-224 and NASA contract NCC-01-001. We also want to thank the McKnight Foundation for their generous support over the years.

Notes

1. NIMBUS is a execution, analysis, and code generation environment for the state-based, fully formal specification language RSML^{-e}.
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.
3. We thank Dr. Steve Miller and Dr. Alan Tribble of Rockwell Collins Inc. for the information on flight control systems and for letting us use the models they have developed in our case studies.

References

Ait-Ameur, Y., Bel, G., Boniol, F., Pairault, S., and Wiels, V. 2003. Robustness analysis of avionics embedded systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*. New York, NY, USA, ACM Press, pp. 123–132.

Biere, A., Cimatti, A., Clarke, E.M., and Zhu, Y. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems*, pp. 193–207.

Choi, Y. 2003. Toward automated verifiaction of software specifications with numeric constraints. Ph.D. thesis, University of Minnesota. Draft.

Choi, Y. and Heimdahl, M. 2002. model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*. Tokyo, Japan, pp. 109–118.

Choi, Y., Rayadurgam, S., and Heimdahl, M. 2001. '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)*, pp. 164–174.

Choi, Y., Rayadurgam, S., and Heimdahl, M.P. 2002. Toward automation for model checking requirement specifications with numeric constraints. *Requirements Engineering Journal*, 7(4):225–242.

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

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pp. 154–169.

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

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. 1991. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320.

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

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

Leveson, N., Reese, J., Koga, S., Pinnel, L., and Sandys, S. 1997. Analyzing Requirements Specifications for Mode Confusion Errors. In *Proceedings of the Workshop on Human Error and System Development*.

Leveson, N.G. and Palmer, E. 1997. Designing automation to reduce operator errors. In *Proceedings of the IEEE Systems, Man, and Cybernetics Conference*.

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

Miller, S.P. and Potts, J.N. 1999. Detecting mode confusion through formalanalysis and modeling. In *NASA Contractor Report NASA/CR-1999-208971*.

Miller, S.P. and Tribble, A.C. 2001. Extending the Four-Variable Model to Bridge the System-Software Gap. In *Proceedings of the Twentith IEEE/AIAA Digital Avionics Systems Conference (DASC'01)*.

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

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252.

Reese, J. and Leveson, N. 1997a. Software deviation analysis. In *International Conference on Software Engineering*.

Reese, J. and Leveson, N. 1997b. Software deviation analysis: A "Safeware" technique. In: *AIChe 31st Annual Loss Prevention Symposium*.

Reese, J.D. 1996. Software deviation analysis. Ph.D. thesis, University of California, Irvine.

Thompson, J.M., Heimdahl, M.P., and Miller, S.P. 1999. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, pp. 163–179.

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

Whalen, M.W. 2005. Trustworthy translation for the requirements state machine language without events, University of Minnesota.