FAULT DIAGNOSIS BASED ON CONTINUOUS SIMULATION MODELS

Stefan Feyock and Randall P. Meyers

Computer Science Department
The College of William and Mary
Williamsburg, Virginia 23185

* The research reported in this paper was sponsored by NASA grant NAG-1-618.

Abstract

Reasoning about physical systems from first principles, whether for purposes of explanation, prediction, or fault diagnosis, requires the reasoner to have available a model of the subject system. A novel approach, based on predicate transformers, to deep reasoning about actual systems on the basis of continuous simulation models of these systems is described. An application to the diagnosis of a malfunctioning Aerobee rocket attitude control system is presented.

1. Introduction

Reasoning about physical systems from first principles, whether for purposes of explanation, prediction, or fault diagnosis, requires the reasoner to have available a model of the subject system: the model, in a sense, is the first principles.

The motivation for the research to be described was the requirement on the part of workers at NASA/Langley Research Center for fault diagnosis techniques based on deep reasoning that could deal with physical systems such as jet engines and avionics subsystems. Among the systems to which our approach has been applied are a buzzer, an Aerobee rocket control system, a relay servo, and an aircraft arresting cable mechanism.

Much of the prior work on fault diagnosis has focused on discrete logical models; most physical systems, however, have a predominantly continuous character and are thus best represented by continuous models. We are interested in systems that are sufficiently complex to be analytically intractable, and must therefore be represented by simulation models. The present paper describes the use of continuous simulation models (CSMs) as the basis of an automated reasoning system, and in particular the application of such a reasoner to fault diagnosis. A novel approach to reasoning on the basis of CSMs was found: the use of Dijkstra's concept of predicate transformers [Dijkstra] to establish weakest preconditions for the model to exhibit the behavior seen in the actual system*.

1.1. CSM-Based Diagnosis

The basis of our approach to model-based fault diagnosis is to observe the actual system in action, and then pose the question: "how do we make the model behave like that?" More precisely, the actual system is not behaving as the simulation predicts; how must the simulation model be changed to exhibit the observed behavior? In the initial phase of this research it was noted that in a large proportion of cases faults corresponded to unexpected (unmodeled) changes in CSM model parameters. In accordance with these observations, we operate under the assumption that variant behavior in the actual system corresponds to such inadvertent parameter variations, rather than to changes in model structure. This assumption causes us to miss some faults, but is surprisingly successful.

* We refer to "actual systems" rather than "physical systems" because CSMs can be used to model a wide variety of entities, including physical, economic, and biological systems.
As an example, consider the case of a linear harmonic oscillator consisting of a weight and a spring. If the spring breaks, it appears plausible that it is necessary to change the structure of the model representing this system, since there has been a radical change both in physical structure and behavior. This is not the case, however; all that is required for the familiar differential equation model to remain valid is to set the spring constant to zero. It was found, in fact, that inability to make a CSM act like a malfunctioning actual system was almost always due not to a requirement for structural changes in the way model components interacted, but rather to aspects of reality that where not represented in the model in the first place. Since any model other than the "identity model" will be less detailed than the actual system, this problem must necessarily remain with us.

Thus, given a set of actual system behaviors, we ask what has to be true in the model domain for the model to behave like that. More formally: what conditions have to hold to produce the observed results?

It is the latter formulation that provides the link to Dijkstra's weakest precondition (wp) approach, which is based on the concept of predicate transformers. Dijkstra's work poses a similar question about programs: given any predicate, the program acts as a predicate transformer that maps the given predicate to the (weakest) predicate that had to be true before the program was executed, in order for the given predicate to hold after execution.

The relevance to our question is clear: we want to know what had to be true (what precondition had to hold) for the actual system to exhibit the observed behavior. The typical actual system - a jet engine, a national economy, a relay servo, an Aerobee rocket, whatever - is, of course, not a program, but its CSM is, and so we can apply the predicate transformer technique to the CSM to derive an answer to the above question.

The situation is unfortunately not that simple: predicate transformers were not developed with CSMs in mind, but rather are oriented toward proving correctness of algorithms. In concise terms, the distinction between a CSM and a non-simulation program is that the latter engages in runtime behavior to produce a result; in a simulation, the runtime behavior is the result. Since simulation results are thus produced by making measurements during the run rather than after the run is complete, it is in general not relevant to ask about postconditions of the simulation loop; after the loop has finished, it is too late. Rather, it is appropriate to inquire what effect a typical iteration of that loop produces.

We now present a summary of predicate transformers, before describing the particular techniques involved in applying this approach to CSMs.

2. Predicate Transformers

A number of special considerations deriving from our application govern our use of predicate transformers, occasioning corresponding differences from Dijkstra's treatment. We are, for example, not interested in questions of program termination; our assumption that the model implementation is correct insures that there are no infinite loops within the loop body. In fact, loops of any sort within the simulation loop body are fairly rare in CSMs, allowing us to minimize confrontations with the most troublesome of the predicate transformers, the one corresponding to the do-od construct.

We present a brief review of the basic predicate transformers; readers requiring more detail are referred to [Dijkstra]. We will use Dijkstra's notation \( \text{wp}(\text{Prog}, R) \) to denote the weakest (most general) predicate, called the weakest precondition, that must hold prior to execution of program \( \text{Prog} \), if predicate \( R \) (the postcondition) is to hold after execution of \( \text{Prog} \). The following transformers are important in the CSM context:

**Statement sequencing**: if \( S_1 \) and \( S_2 \) are program statements, then

\[
\text{wp}(S_1; S_2, R) = \text{wp}(S_1, \text{wp}(S_2, R)).
\]

This fact immediately generalizes to

\[
\text{wp}(S_1; \ldots; S_n, R) = \text{wp}(S_1, \text{wp}(S_2, \ldots, \text{wp}(S_n, R) \ldots)).
\]

**Assignment**: \( \text{wp}(x := E, R) = \text{R}_{E \rightarrow x} \)

where \( \text{R}_{E \rightarrow x} \) denotes predicate \( R \), with expression \( E \) substituted for all free occurrences of variable \( x \) in \( R \). For example,

\[
\text{wp}(x := 1 - y, x = y + 5) = y + 5 = 1 - y, \text{or } y = -2.
\]
which is the weakest predicate that had to be true if the postcondition \( x = y + 5 \) was to hold.

The \texttt{if} statement has the usual guarded-command syntax:

\[
\texttt{<if> ::= if <boolean-exp> -> <statements> ;}
\]

\[
\texttt{<boolean-exp> -> <statements> fi}
\]

The transformer for \texttt{<if>} is

\[
\text{wp(<if>, R) = (if <boolean-exp1> then wp(<statements1>, R)) AND}
\]

\[
\text{AND}
\]

\[
(\text{if <boolean-exp}_p> then wp(<statements>_p, R))
\]

In our application we can state that exactly one guard will be true. The condition

\[
(\text{<boolean-exp1> OR --- OR <boolean-exp}_p>)
\]

included in Dijkstra's formulation is therefore always true and can be omitted.

3. Continuous Simulation Models

It is important to note that the term "continuous simulation model" (CSM) is misleading, having as its opposite not "discontinuous simulation model", but rather "discrete-event simulation model". CSMs can, and usually do, have arbitrarily discontinuous aspects; it is the intractability resulting from discontinuities as well as other sources that prompts the use of simulation rather than analysis.

3.1. Continuous Simulation: Schematic Form

A continuous simulation of the sort we shall discuss proceeds by integrating a system of ordinary differential equations. The approach we shall take is based on Euler integration, which derives from the following straightforward principles:

1. The progression of time is quantized into discrete steps of size \( dt \). The smaller the value of \( dt \), the more accurate (and slow) the simulation becomes; \( dt \) is not, however, deemed to be infinitesimal.

2. distance = rate \(*\) time. In this context, "distance" refers to the amount a variable changes from one time step to the next "rate" is given in terms of differential equations determining the variable; "time" is, of course, \( dt \).

The schematic form of a continuous simulation has the following structure:

\[
\text{loop}
\]

A. use values of simulation variables at time \( t \) to compute, by means of model equations, time \( t \) values for the highest derivatives occurring in the model

B. use time \( t \) values of derivatives and other variables to compute, by means of the \( d = \text{rate} \times \text{time} \) principle, time \( t+1 \) values for all but the highest derivatives:

\[
\text{C. } t := t + 1
\]

4. An Aerobee Control System Simulation

We will present our technique in light of an example, taken from [Chu], that simulates the control system of an Aerobee rocket. This simulation is complicated enough to be non-trivial, and concise enough to remain comprehensible, at least until the computer gets its hands on it.

The Aerobee is a small research rocket used to carry scientific payloads into space. It contains an attitude control system which can be used to orient instruments toward designated celestial objects. This control system utilizes two reference and three orthogonal rate gyros to determine orientation and rates of change of attitude. For simplicity, only a single-axis system will be simulated.

Space restrictions preclude a detailed description of the attitude control system components and circuitry; the reader is referred to [Chu] for further discussion. In outline form, the system operates as follows. The free gyro produces an error voltage \( EV = G2 \times \sin(\text{RA} - \text{X}) \), where \( G2 \) is the gyro sensitivity, \( \text{RA} \) the reference angle (i.e., the intended angle of orientation), and \( \text{X} \) is the actual angle of orientation of the rocket. This error voltage \( EV \) is added with a feedback voltage \( FV = -G1 \times \text{X}' \) from a rate gyro by a mixing network that contains resistors \( R1, R2, \) and \( R3 \); \( G1 \) is the sensitivity of the rate gyro. The voltage \( V \) output by

150
the network is given by $V = C_1 EV + C_2 FV$, where

$$C_1 = \frac{1}{1+(1+R_3/R_2)\times(R_1/R_3)}$$

and

$$C_2 = \frac{1}{1+(1+R_3/R_1)\times(R_2/R_3)}$$

The dynamics of the rocket can be taken as $X' = FMA/I$, where $M$ is the moment arm, $I$ is the moment of inertia of the rocket about its longitudinal axis, and $F$ is the force produced by the gas thrusters used to control the angular orientation. These thrusters can produce only three discrete levels of forces: $FA$, $0$, and $-FA$; there is a dead space of $2\times DS$.

The following LISP statements represent a simulation of this nonlinear feedback control system. Note that the program statements are in infix form; for reasons extraneous to this discussion it was considered desirable to implement an infix-to-prefix parser to preprocess such statements.

```
(value assignments to system parameters and constants
(setq C1 '(ra := 0)) ; reference angle
(setq C2 '(i := 900)) ; moment of inertia
(setq C3 '(ma := 12.5)) ; moment arm
(setq C4 '(dt := .01)) ; time step size
(setq C5 '(gl := 0.13)) ; rate gyro sensitivity
(setq C6 '(g2 := 11.9)) ; free gyro sensitivity
(setq C7 '(ds := 0.025)) ; dead space in thruster controller
(setq C8 '(r1 := 33000)) ; resistance of R1 in mixing net
(setq C9 '(r2 := 33000)) ; resistance of R2 in mixing net
(setq C10 '(r3 := 25000)) ; resistance of R3 in mixing net
(setq C11 '(pi := 3.14159))
(setq C12 '(x := 12.08)) ; orientation of rocket
(setq C13 '(x1dot := -1.27)) ; $x'$, the first derivative of $x$
(setq C14 '(rc := pi / 180)) ; radian conversion factor
(setq C15 '(dc := 180 / pi)) ; degree conversion factor
(setq M1 '(c1 := 1/(1+(1+r3/r2)\times(r1/r3))) ; mixing network
(setq M2 '(c2 := 1/(1+(1+r3/r1)\times(r2/r3))) ; coefficients
 ; statements:
(setq S1 '(ev := g2 * sin((ra - x) * rc)) ; error voltage
(setq S2 '(fv := NEG g1 \times x1dot)) ; feedback voltage
(setq S3 '(v := c1 \times ev + c2 \times fv)) ; network output voltage
(setq S4 'IF
    ((v < NEG ds) => (h := v + ds))
    ((v <= ds) => (h := 0))
    ((ds < v) => (h := v - ds))
)) ; thruster controller dead space
(setq S5 'IF
    ((h < 0) => (f := -4))
    ((h = 0) => (f := 0))
    ((0 < h) => (h := 4))
)) ; thruster force
(setq S6 '(x2dot := (f \times ma / i) \times dc)) ; rocket dynamics
 ; part B of the simulation: updating of $x$ and $x'$:
(setq S7 '(x := x + x1dot \times dt))
(setq S8 '(x1dot := x1dot + x2dot \times dt))
```

The above program statements constitute parts A and B of the simulation loop for the Aerobee attitude controller. We now illustrate how the above simulation model can be used for fault diagnosis. Suppose that in the actual rocket resistor $R_1$ burns out. Telemetry transmits a sequence of data corresponding to the rocket's subsequent behavior, including values of angle $x$. From the sequence of $x$ values we can derive
first and second derivatives $x'$ and $x''$ by differencing. Presumably (since our rocket is tumbling) these values reflect a pathological condition; the question, as usual, is "how do we make the model act that way?"

It is not immediately obvious what is meant by "that way"; to be completely general, we presumably need to make the model reproduce the entire set of data observed. To determine values of system parameters that will produce such output, however, it will not, in general, be necessary to examine the entire set of data. Rather, it suffices to examine a "pathological subset", i.e. a subset of the data that could not have been produced by a normally operating system. The size of this subset depends on the structure of the simulation. As it happens, a single triple of values of $x$, $x'$, $x''$, chosen from "pathological" triples differing from triples representing normal behavior, is sufficient to produce a diagnosis in this case. This is fortuitous and specific to the R1 fault; had we not been so lucky, we would have had to extend our weakest-precondition analysis to include additional data points.

The analysis proceeds by posing the question "how can the model be made to produce the chosen triple of values, i.e. $x = 12.08$, $x' = -1.27$, and $x'' = 3.18$?" More precisely, since $x''$ is computed from $x$ and $x'$ values, what is the weakest precondition for the model to produce the value $x'' = 3.18$, given that $x = 12.08$ and $x' = -1.27$? We thus incorporate the first two conditions into the program as assignment statements. We then have

\[
\text{wp (prog '}\text{x2dot = 3.18})\]

where prog is defined by

\[
\text{setq prog (list C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 M1 M2 S1 S2 S3 S4 S5 S6))}
\]

The remaining statements of the loop are omitted, since the question is how a value of $x'' = 3.18$ could be computed, given values of 12.08 and -1.27 for $x$ and $x'$, and it is the included statements that perform this calculation. Cases where the entire loop body must be included exist but are beyond the scope of this paper.

As its name suggests, wp is the function in our LISP implementation of predicate transformers that performs the weakest precondition analysis. prog is not a huge program; nonetheless, the wp analysis soon reaches the point of intractability for the human analyst. The result of invoking wp with the above arguments is a boolean expression composed arithmetic expressions, comparison operators, and connectives such as AND and OR, whose length is nearly a page; space limitations unfortunately preclude display of this monstrosity. Clearly an inequality of this size is intractable in human terms; an automatic simplification program was therefore implemented. Application of this program, however, shows that this complicated expression evaluates to false! The meaning of this result is that given the above parameter values, it is not possible for the program to produce an $x''$ value of 3.18. We therefore have a malfunction: at least one of the parameters does not have the value we believe it to have. But which one? The procedure we have adopted is to cycle through the parameters, considering each in turn to be suspect.

More precisely, we pick a parameter, and run the wp analysis with no assumption made about that one parameter's binding, a process we call disengaging the parameter. Two outcomes are possible: the resulting weakest precondition is still false, indicating that the suspect parameter is innocent. Alternatively, the weakest precondition imposes a value, or range of values, which the suspect parameter must have in order for the postcondition ($x'' = 3.18$, here) to hold. If the value constraint imposed by the weakest precondition indicates that the parameter has to be something other than its supposed normal value in order for the postcondition to hold, that parameter is strongly suspect.

It must be emphasized that the above procedure is heuristic. It is possible, for example, for the normal values of all parameters to be consistent with the wp constraints. This can happen if not enough global information was supplied. It is also possible for the parameter value demanded by the weakest precondition to be "off", even though the parameter is in fact "innocent", i.e. the actual malfunction is elsewhere. If, for example, the malfunction is R1 burnout, and we disengage R2, it may happen that there is a possible value for R2 that produces the observed behavior. This difficulty can be minimized by using domain-specific knowledge about the way components can fail. For example, if the wp value of R2 is negative (as in fact proves to be the case in our example), we can clear R2 of suspicion, since resistances cannot be negative. On the contrary, the most probable failure mode of a resistor is to burn out, which has the effect
of greatly increasing its resistance, possibly to infinity.

In accordance with the above discussion, the computer wp analysis of the Aerobee model was performed with R1 disengaged. Note that we disengage a variable by simply failing to assign it a value in the program. The result is once again a boolean expression extending over nearly a page. To convey a flavor of the result, here are the final six lines (out of 30) of this expression:

\[ \begin{align*}
&11.9 \sin((0-12.08) * (3.14159/180)) + (1/(1+(1+25000/r1) * 33000/25000)) * \\
&(\neg .13 * -1.27)) -.025) = 0) \lor \\
&(12.5/900) * (180/\pi)) = 3.183) \land (379649.5546, \infty),
\end{align*} \]

As is generally the case, this expression consists of constants together with the disengaged variable. As is also generally the case, the expression is too complicated to be solved analytically, even after the expressions are evaluated as far as possible; numerical methods were therefore applied. The approach we have implemented produces a set of intervals such that the boolean expression is true iff the value of the disengaged variable lies in one of the intervals. The solution obtained for r1 was the interval [379649.5546, infinity), indicating that the value of r1 was at least an order of magnitude higher than the nominal value of 33000 ohms. This condition, together with the results of disengaging the other parameters, strongly suggests that R1 is burned out.

5. Conclusion
We have described a novel method of deep reasoning about the behavior of systems based on descriptions of these systems in terms of continuous simulation models. Space limitations have confined us to a brief overview of our approach here; the reader is referred to [Feyock] for details. As was evident from the discussion, the models furnished to the reasoner were not based on if-then rules or any other sort of qualitative logical model, but rather were quantitative CSMs of the actual systems. This fact has a number of interesting implications, including the integration of qualitative reasoning with quantitative simulation, and obviating the need to develop a qualitative model from the mathematical description of the system.

The approach we have described has had a number of encouraging successes, but is easily overwhelmed by larger simulations. We have found that applying our approach to an envisionment of the system rather than to the CSM itself has greatly enhanced the power of this method. A procedure for automatically generating such an envisionment from a CSM, similar in spirit to the work of [deKleer], has been developed and implemented, and its application in conjunction with the predicate transformer approach is currently under investigation.

BIBLIOGRAPHY


