Title of Grant:

FAULT DIAGNOSIS BASED ON CONTINUOUS SIMULATION MODELS

Grant Number: NAG-1-618

Period of Grant: October 1, 1986 - December 31, 1987

Principal Investigator: Stefan Feyock

Grantee Institution:

The College of William & Mary
Williamsburg, Virginia 23185

FINAL REPORT

FAULT DIAGNOSIS BASED ON CONTINUOUS SIMULATION MODELS

Final Report
NASA/Langley Grant NAG-1-618

Stefan Feyock
Department of Computer Science
College of William & Mary

October 28, 1988

Abstract

~~The present report describes~~ the results of an investigation of techniques for using continuous simulation models as basis for reasoning about physical systems, with emphasis on the diagnosis of system faults. It is assumed that a continuous simulation model of the properly operating system is available. Malfunctions are diagnosed by posing the question: "how can we make the model behave like that?" The adjustments that must be made to the model to produce the observed behavior usually provide definitive clues to the nature of the malfunction. A novel application of Dijkstra's weakest-precondition predicate transformer is used to derive the preconditions for producing the required model behavior. To minimize the size of the search space, an envisionment generator based on interval mathematics was developed. In addition to its intended application, the ability to generate qualitative state spaces automatically from quantitative simulations proved to be a fruitful avenue of investigation in its own right. Implementations of the Dijkstra transform and the envisionment generator are reproduced in the Appendix.

Contents

INTRODUCTION

The present report describes the results of a research project extending over two years, consisting of an investigation of techniques for using continuous simulation models (CSMs) as basis for reasoning about physical systems. In particular, techniques for model-based reasoning about faults of physical systems were to be investigated. The underlying idea is as follows: we assume that a CSM of the actual system is available, and that this model reflects the behavior of the actual system with a fidelity that suffices for the given application. A malfunction in the actual system will produce *symptoms*, i.e. a stream of data (observations) reflecting the aberrant behavior. We then pose the question "how can we make the model act like that?" Presumably the adjustments that must be made to the model to produce the observed behavior will provide definitive clues to the nature of the malfunction.

Such an approach is based on two important assumptions. The first is that the model on which the diagnosis is based is a *component model* rather than a response surface, i.e. that the structure of the model reflects the nature of the actual system. (The research area known as *causal modeling* is concerned with this same issue, and in fact carries these concerns further than our application requires.) By way of example, consider the familiar linear harmonic oscillator, described by the equation

$$m * x'' + d * x' + k * x = F(t)$$

This equation is a constraint model rather than a causal model. Nonetheless, attributes of the basic system components *are* represented; for example, if we are dealing with a mass-spring system, then m represents the mass, d the damper resistance, k the spring stiffness, and $F(t)$ the driving force. By contrast, a response surface model may be constructed by fitting, say, a polynomial to data representing the behavior of a mass- spring system. Such a model will in no way reflect the structure of the underlying system, and will be unsuitable for system identification-based diagnosis.

The second assumption underlying our approach is that the adjustments to the model that make it reproduce the aberrant behavior observed in the actual system consist of finding new values for system parameters, rather than structural changes such as addition of new equations. This is plausible, in that

addition of new structure in the form of equations corresponds to addition of new constraints. Malfunctions, however, generally involve the modification or removal of *existing* constraints, which can frequently be modeled by parameter adjustments. For example, spring breakage in the mass-spring system discussed above can be represented by setting k to 0.

Three innovations which we consider to be significant were developed and integrated. They are:

1. the development of a canonical schematic form for continuous simulation models

2. the application of Dijkstra's predicate transformers to algorithmic mappings, leading to techniques and results with extend techniques familiar from continuous mathematics to CSMs

3. the development of qualitative reasoning techniques based on quantitative models. These techniques include the application of interval mathematics to perform automatic generation of envisionments from CSMs, and the use of predicate transformers to do fault diagnosis on the basis of these envisionments.

In the subsequent discussion we will often use the abbbreviations *qn* for "quantitative" and *ql* for "qualitative", since these are more easily distinguished than their abbreviands.

The results we have developed afford a number of interesting analogies with techniques from continuous mathematics, particularly dynamical systems theory [4]. Among these are formal parallels in model structure, the use of transforms to derive information from the model, and conceptual similarities in the treatment of questions such as steady-state conditions and system identification. We have even found a transform-based represention of iterative programs that is formally analogous to a power series expansion.

While the results to be described are mathematically interesting, the intent of this research was to develop approaches to the diagnosis of faults in actual physical systems. We believe our results to be of practical value for several reasons:

- CSMs of physical (and frequently economic, urban, and other) systems are often available. An understanding of the distinct operating regions of such models is almost always necessary in order to reason about the system. An envisionment is a schematic representation of these operating regions; it is therefore evident that a technique for automatic generation of envisionments from CSMs should be of value.

- The Dijkstra transform, which was originally developed in a programming methodology/software engineering context, has proved to be a powerful tool for pre-and postdiction when applied to CSMs. Chapter 1 of this

report establishes its general usefulness by describing its application to derivation of steady-state conditions; in Chapter 3 the transform is applied to our particular research area: fault diagnosis.

In order to test the techniques which were developed, LISP implementations of the Dijkstra transform, interval mathematics operations, and the interval mathematics-based envisionment generator were written (see Appendix). The resulting programs currently reside on a MacIntosh^{tm} personal computer; we consider their satisfactory performance on such a small machine to be evidence of the practical feasibility of the underlying ideas.

Chapter 1

Models of Physical Systems

The title of this section notwithstanding, the systems under consideration here include non-physical types such as economic, biological, and urban dynamics systems, among others. The only requirement for consideration is that the system be describable in terms of relationships among a finite set of model variables and their rates of change. For the purposes of this research we will, in fact, be interested chiefly in physical systems, particularly systems associated with aviation and avionics.

1.1 Dynamical systems

When the previously mentioned relationships among model variables and their derivatives can be expressed in terms of algebraic equations, then the model takes the form of a system of differential equations. In the course of our investigation it became evident that since the models with which we are concerned usually represent physical systems, additional structure can be imposed on the system of equations. This structure is derived from considerations of *dynamical systems theory*, which affords a device-oriented ontology as well as a unifying and methodical approach to the modeling of physical systems. In this approach, physical systems are deemed to be made up of only seven basic component types: generalized *capacitors*, which store "effort", generalized *inductances*, which store "flow", generalized *resistances*, which dissipate energy, transformers and gyrators, which model transmissions, and effort and flow sources. Two junction types suffice to connect these components. The interested reader is urged to consult a work such as [4] for a detailed presentation of these ideas, since this elegant and highly effective ontology diverges from those customarily employed in qualitative reasoning research. For our purposes, the importance of dynamical systems theory lies in the fact that it provides a systematic approach by

which any model of a physical system can be put in the form

$$\begin{aligned} x_1' &= F_1(x_1, \cdots, x_n, u_1, \cdots, u_r) \\ &\cdots \\ x_n' &= F_n(x_1, \cdots, x_n, u_1, \cdots, u_r) \end{aligned}$$

or, in vector form,

$$\vec{X}' = \vec{F}(\vec{X}, \vec{U})$$

where $x_i, i = 1, \cdots, n$, represents the amount of effort or flow stored in (capacitive or inductive) component i, and $u_j, j = 1, \cdots, r$, is an *input variable*. The x_i are called the *state variables* of the system; the above form of system equations is accordingly called *state variable form*. The F_i represent algebraic functions, i.e. compositions of elementary functions; we will therefore refer to functions such as $\vec{F}$ as *algebraic mappings*. It can in fact be shown that

$$F_i(x_1, \cdots, x_n, u_1, \cdots, u_r)$$

may be assumed to have the form

$$g_{i1}(x_1) + \cdots + g_{in}(x_n) + h_{i1}(u_1) + \cdots + h_{ir}(u_r)$$

since the process of equation derivation [4] produces such a form. A linear system is then a special case where $g_{ik}(x_k) = \pm a_{ik}x_k$ and $h_{ir}(u_r) = \pm b_{ir}u_r$.

It should also be pointed out that it is in general a trivial matter to reduce n-th order ($n > 1$) differential equations to systems of n first-order equations. We need merely introduce intermediate variables in an appropriate manner. For example, in the case of a mass-spring system defined by the equation

$$m * x'' + d * x' + k * x = F(t)$$

we define a new variable $y = x'$. The second-order equation then becomes the system of two first-order equations

$$\begin{aligned} x' &= y \\ y' &= -(1/m) * d * y - (1/m) * K * x - (1/m) * F(t) \end{aligned}$$

The resulting system has the advertised form

$$\vec{X}' = \vec{F}(\vec{X}, \vec{U})$$

but is unsatisfactory insofar as the X and U are not in general the actual energy variables and inputs, but linear transforms of these.

In terms of our mass-spring example, the capacitative element is the spring, which stores effort (generalized force), the inductive element is the mass, which

stores flow (generalized velocity), and the dissipative element is the damper, which acts as a resistance. The state variable form of

$$m * x'' + d * x' + k * x = F(t)$$

is

$$\begin{aligned} x' &= -kq + dx/m - F(t)/m \\ q' &= x/m \end{aligned}$$

where q is momentum, x displacement. [4] should be consulted for details of the process of constructing systems of equations.

Note that in this case the state function $\vec{F}(\vec{X}, \vec{U})$ is *linear*, i.e. it has the form

$$\begin{aligned} x_1' &= a_{11}x_1 + \cdots + a_{1n}x_n + b_{11}u_1 + \cdots + b_{1r}u_r \\ &\vdots \\ x_n' &= a_{n1}x_1 + \cdots + a_{nn}x_n + b_{n1}u_1 + \cdots + b_{nr}u_r \end{aligned}$$

or, in vector form, $\vec{X}' = A\vec{X} + B\vec{U}$, where A, B are coefficient matrices.

When linearity obtains, an extensive arsenal of powerful mathematical techniques can be brought to bear, to such an extent that non-linear $\vec{F}$ functions are often made tractable by replacing them with linear approximations. Among these tools are Laplace transforms, eigenvalues, poles and zeroes, and numerous other techniques for establishing steady-state conditions, oscillatory modes, and similar system properties.

The state variable formulation of the linear harmonic oscillator exemplified by the mass-spring system above yields a simple example. The system is stable if none of the variable values are changing, i.e. if $p' = 0$ and $x' = 0$. This happens when

$$\begin{aligned} 0 &= -kq + dx/m - F(t)/m \\ 0 &= x/m \end{aligned}$$

Solving these equations, we see that the system is stable iff $x = 0$, and $F(t) = -qm$. More generally, we can obtain the equilibrium (steady-state) conditions for an arbitrary system $\vec{X}' = A\vec{X} + B\vec{U}$ by solving the system of simultaneous equations $\vec{0} = A\vec{X} + B\vec{U}$ for $\vec{X}$ in the usual manner.

1.2 CSMs of Intractable Systems

Instead of assuming models with docile properties such as linearity, our research has taken the opposite approach, concentrating on discontinuous, non-linear,

analytically intractable systems, since these are the ones for which simulation rather than analysis is appropriate. Rather than assuming that $\vec{F}$ can feasibly be linearized, or even that it is a continuous, differentiable function of its arguments, we do not require $\vec{F}$ to be a closed-form function at all, but rather allow it to denote an arbitrary *algorithm*, possibly including assignments to temporaries, **if-fi** statements, and explicit dependence on time (as in, say, **if** $t \geq 4$ **then** $x := 48$ **else** $x := 0$), for computing the indicated values. It should be noted that the term "continuous simulation" is quite misleading: the F_i can be discontinuous, non-linear, non-analytical, and in fact should not be taken literally as closed-form functions at all: the notation F_i merely denotes the fact that x_i' is computed from other values on the basis of some *arbitrary* computation. The term "continuous simulation" has as its opposite "discrete-event simulation", not "discontinuous simulation". To emphasize this fact, we shall generally use the notation $\mathcal{A}(\vec{X}, \vec{U}; t)$ instead of $\vec{F}(\vec{X}, \vec{U})$ to emphasize the algorithmic nature of the computation.

1.2.1 Continuous Simulation: Schematic Form

As indicated, we are interested in a particular kind of model called a *Continuous Simulation Model*, which is appropriate if the system in question can be described in terms of relationships among a set of model variables and their derivatives, but the relationships are mathematically intractable.

A continuous simulation of the sort we shall discuss proceeds by integrating a system of ordinary differential equations. Since a digital computer is a discrete machine, this process must necessarily proceed in discrete steps. 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. The relation $distance = rate * time$. 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:

```
initialize all but highest derivatives
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 tocompute, by means of the d = r*t
           principle, time t+1 values for all but
           the highest derivatives
    end loop;
```

Let us examine this computation schema in more detail. Suppose the variables involved in the simulation are $x_1, \ldots, x_n$, as well as their derivatives $x_i^{(1)}, \ldots, x_i^{(m_i)}$. (Note: we will use the term "highest derivative(s)" for the highest-order derivative of each variable occurring in the model, and "lower derivatives" to denote all derivatives other than the highest derivatives. Included in these lower derivatives is the zero-order derivative of each variable, i.e. the variable itself: $x_i^{(0)} = x_i$.) Also given are functions relating the highest derivative of each variable to the values of the lower derivatives and of auxiliary variables:

$$\begin{aligned} x_1^{(m_1)} &:= \mathcal{A}_1(\langle args_1 \rangle) \\ &\vdots \\ x_n^{(m_n)} &:= \mathcal{A}_n(\langle args_n \rangle) \end{aligned}$$

The arguments $\langle args_i \rangle$ typically include lower derivatives (including the variables themselves), as well as auxiliary variables. Inclusion of highest derivatives in the arguments is permissible, but must be handled carefully to avoid circularity. As is the case in the analytical approach, a set of initial conditions is given, which specify initial values of $x_i^{(j)}, j = 0, \ldots, m_j - 1$. We thus begin with known values for all except the highest derivatives $x_1^{(m_1)}, \ldots, x_n^{(m_n)}$. The first step is to complete this set by computing the highest derivatives for dt :

$$\begin{aligned} x_1^{(m_1)}[0] &= \mathcal{A}_1(\langle args_1 \rangle) \\ &\vdots \\ x_n^{(m_n)}[0] &= \mathcal{A}_n(\langle args_n \rangle) \end{aligned}$$

(the notation $x_i^{(j)}[t]$ denotes the value of $x_i^{(j)}$ at time t). At this point we have values for all $x_i^{(j)}, i = 1, \ldots, n, j = 0, \ldots, m_j$. We can now use these values and the $d = r \cdot t$ principle to compute values for time $t+1$. Since t was 0 above, the next step computes $t = 1$ values:

$$\begin{aligned} x_1[1] &:= x_1[0] + x_1' \cdot dt \\ &\quad -- \; z' denotes \; dz/dt \\ x_1'[1] &:= x_1'[0] + x_1'' * dt \\ &\vdots \\ x_1^{(m_1 - 1}[1] &:= x_1^{(m_1 - 1}[0] + x_1^{(m_1)}[0] \cdot dt \end{aligned}$$

From this we see that the general computation proceeds as follows:

loop
- at this point we have time t values for
- all but the highest derivatives

$$x_1^{(m_1)}[t] := f_1(\langle args_1 \rangle)$$
$$\vdots$$
$$x_n^{(m_n)}[t] := f_n(\langle args_n \rangle)$$

- at this point we have time t values of *all* variables
- and their derivatives.
- Now compute time $t+1$ values for all but the
- highest derivatives

$$x_1[t+1] := x_1[t] + x_1'[t] * dt$$
$$\vdots$$
$$x_1^{(m_1-1)}[t+1] := x_1^{(m_1-1}[t] + x_1^{(m_1)}[t] * dt$$
$$\vdots$$
$$x_n^{(m_n-1)}[t+1] := x_n^{(m_n-1)}[t] + x_n^{(m_n)}[t] * dt$$

- now update t and iterate

$t := t+1;$
end loop;

The $x_i^{(j)}$ above were treated as array variables indexed by t. In general, however, it is not necessary to save all values of all $x_i^{(j)}$ for all t; the computations typically proceed quite locally in time. The loop can thus be rewritten as follows:

loop
- point A:

$$x_1^{(m_1)} := f_1(\langle args_1 \rangle)$$
$$\vdots$$
$$x_n^{(m_n)} := f_n(\langle args_n \rangle)$$

- as before, we now have a full set of values for
- - $x_i^{(j)}, i = 1, \ldots, n, j = 0, \ldots, m_j$
- point B:
- Now compute time $t+1$ values:

$$x_1 := x_1 + x_1'[t] * dt$$
$$x_1' := x_1' + x_1''[t] * dt$$
$$\vdots$$
$$x_1^{(m_1-1)} := x_1^{(m_1-1)} + x_1^{(m_1)} * dt$$
$$\vdots$$
$$x_n^{(m_n-1)} := x_n^{(m_n-1)} + x_n^{(m_n)} * dt$$

```
  - explicit updating of t is no longer necessary
  - point A:
end loop;
```

Note that we have designated two control points in the above program; we call them point A and point B (the point A just before the **end loop** is logically identical with the point A following **loop**). Moreover, we shall refer to the computations following point A and preceding point B as the *A-computations* or the *A block*; similarly, the block of code after B and before A is called the *B-computations* or the *B block*. A number of observations are in order regarding the above program. The first is that while the order of the B computations is irrelevant if explicit time subscripts are used, order is critical if, as above, subscripts are omitted. Suppose, for example, that we had written

$$\begin{aligned} x_1' &:= x_1' + x_1'' * dt \\ x_1 &:= x_1 + x_1' * dt \end{aligned}$$

It is clear that the x_1' used in the second equation to update x_1 is $x_1'[t+1]$, not $x_1'[t]$. As noted previously, the notation f_i merely denotes the fact that $x_i^{(m_i)}$ is computed from other values on the basis of some *arbitrary* computation, which may involve loops, **if** statements, procedure calls, and all the other mechanisms of computation.

We now present an example of such a simulation: a program which models a relay servo. (Source: [1], pp. 117 - 119.) A relay servo is a feedback control system in which the corrective signal is applied discontinuously. The intent of the system is to minimize the error difference $E = Y - X$ between reference (input) signal Y and the output X (Y is kept constant at zero in this model). The servo equation is

$$X'' = -X'/B + G \cdot A/B$$

where G represents the action of the relay, taking on values of -1, 0, and $+1$, depending on the value of E. The formal similarity of the above equation with the linear harmonic oscillator equation $x'' = -x'/mass - k \cdot x/mass$ should be noted; A plays the role of the spring constant, B the role of mass.

```
 -- parameter specifications:
A := 2; b := 0.5; V := 1; Y := 0;
 -- step size specification:
dt := 0.01;
 -- initial conditions:
X' := 0; X := 1.5;
 -- simulation loop:
loop
 -- point A:
  E := Y - X;
```

```
  if E > 0 then G := V;
  elsif E = 0 then G := 0;
  else G := -V;
  X'' := -X'/B + G * A/B
   -- X''[t] = f(X'[t], G[t], A, B)
   -- All values X, X', X'' are current to time t here
   -- Now compute t+1 values of X and X':
 -- point B:
  X := X +  X' * dt
  X' := X' + X'' * dt
end loop
```

1.3 Components of a Continuous Simulation Model

The previous section has presented a schematic form for CSMs. We now examine the constituents of CSMs more closely. These may be classified as follows:

- Constants: quantities that cannot change, even in principle (in particular: in the presence of faults).

  Example: $conversion\_factor = 180/3.14159$

- Parameters: quantities treated as constants for the duration of a simulation run, but which may vary from one run to the next.

  Example: $mass = 2.0$

  In the context of fault diagnosis and fault propagation, parameters fall into two classes: intentional parameters (those intended by the programmer to be parameters) and inadvertent parameters, i.e. quantities intended by the programmer to remain constant, but which have changed due to a fault. Inadvertent parameters can, in fact, act as *variables*, changing in mid-run.

- Endogenous variables: $x_1, \ldots, x_n$, and their derivatives: the quantities whose values are determined by the equations of the model. In the case of dynamical systems in state variable form, endogenous variables correspond to state variables. In the relay servo example (which is not in state variable form), $\vec{X}$, $\vec{X}'$, and $\vec{X}''$ are endogenous variables.

- Exogenous variables: quantities supplied (input) from outside the model, rather than being computed in terms of model quantities. In the relay servo example, the reference signal Y is an exogenous variable (which is held constant at zero in this particular example). In dynamical systems, exogenous variables are referred to as *input variables* or *sources* of effort or flow.

- Auxiliary variables: quantities computed in terms of endogenous and exogenous variables and other auxiliary variables, but which are not themselves endogenous variables. In the relay servo example G and E are auxiliary variables. It should be noted that auxiliary variables occur in algorithmic mappings but not in equational ones: only algorithms require temporary variables.

In an algorithmic model constructed without the benefit of a methodology such as the one outlined in [4], the difference between auxiliary variables and endogenous variables is in general subjective and arbitrary. As a rule of thumb, a variable is deemed to be endogenous if it is the sort of thing that might reasonably be printed out as model output, otherwise it is auxiliary. In a system dynamics model, the category of any variable is evident: if it represents an effort or flow source, it is an input (exogenous) variable; if it represents the amount of effort or flow stored in a capacitance or inductance, it is a state (endogenous) variable; if it belongs to neither class, the only remaining possibility is that it is an auxiliary variable.

1.3.1 State Variable Form

While we will continue to deal with models involving higher-order equations, we have seen that we may confine ourselves to systems of first-order equations if desired. In this case we can restate our CSM loop schematic as follows:

```
Assign initial values to simulation variables
loop
  A. use values of simulation variables at time t to compute,
     by means of model equations, time t values for the
     derivatives occurring in the model
     (which will be first-order only)

  B. use time t values of derivatives and other variables to
     compute, by means of the d = r*t principle, time t+1 values
     for all variables
end loop;
```

or in vector form,

$\vec{X} := \vec{X}_0$
loop
  A. $\vec{X}' := \mathcal{A}(\vec{X}, \vec{U}, t)$
  B. $\vec{X} := \vec{X} + \vec{X}' \cdot dt$
end loop

In our previous discussion of dynamical systems models having the general form $\vec{X}' = \vec{F}(\vec{X}, \vec{U}, t)$, we referred to the mapping F as an *algebraic* or *equa-*

tional mapping. By analogy, we refer to $\mathcal{A}$ as an *algorithmic mapping.* More generally, an algorithmic mapping is any computational scheme that maps a set of input values X to a set of output values Y.

As is apparent from the loop schema, the simulation loop body consists of two consecutive mappings, the A-computations $\mathcal{A}$ followed by the B-computations $\mathcal{B}$. The effect of running the simulation for n iterations can then be expressed as applying the mapping $(\mathcal{A} \cdot \mathcal{B})^n$ to argument X_0. If the loop has the form

loop
  A. $\vec{X}' := \mathcal{A}(\vec{\mathcal{X}}, \vec{U}, \sqcup)$
  B. $\vec{X} := \vec{X} + \vec{X}' \cdot dt$
end loop

then we see that initially $\vec{X}_i = \vec{X}_0$, and for $i \geq 0$ we have

$$\vec{X}_{i+1} := \vec{X}_i + dt \cdot \mathcal{A}(\vec{X}_i, \vec{U}_i, t)$$

It is evident that each iteration of the simulation loop computes the next value of $\vec{X}$ by adding a small vector pointing in the $\mathcal{A}(\vec{X}, \vec{U}, t)$ direction to the current $\vec{X}$. It is fairly common for simulations to have neither exogenous variables nor explicit time dependences; if this is so, then the above assignment becomes

$$\vec{X}_{i+1} := \vec{X}_i + \mathcal{A}(\vec{X}_i) \cdot dt$$

The significance of this form is that the behavior of $\vec{X}$ under the iteration is governed by the *time-invariant* mapping $\mathcal{A}$; in particular, the fact that the mapping does not depend on t makes it possible to pre-establish operating regions in the vector space. Such operating regions will play a role in the subsequent development; in particular, each *qualitative state* corresponds to a distinct operating region.

Chapter 2

CSM-based reasoning

We have presented a formalization of the CSM process in terms of iterated mappings on a vector space R^n. We will now explore some of the implications of this formulation.

We have indicated previously that the system identification approach to diagnosis consisted of posing the question "how do we make the model behave like that?" As it happens, this approach is not confined to either fault diagnosis or qualitative reasoning. Consider a dynamical system $\vec{X}' = \vec{F}(\vec{X}, \vec{U})$, and suppose we are interested in steady-state (equilibrium) conditions, i.e. in conditions under which the system is stationary. In model terms, this corresponds to having no changes in any of the state variables; formally, $\vec{X}' = \vec{0} = \vec{F}(\vec{X}, \vec{U})$. The problem has been reduced to finding values for $\vec{X}$ and $\vec{U}$ satisfying $\vec{0} = \vec{F}(\vec{X}, \vec{U})$, which is generally possible, at least numerically, if the system equations are closed-form expressions. In the case of linear systems the solution process reduces to solving a set of linear equations.

The case where the system model is a CSM rather than a dynamical system model presents rather greater problems. Instead of a system of equations $\vec{X}' = \vec{F}(\vec{X}, \vec{U})$, we have an interated computation

loop
$\vec{X}' := \mathcal{A}(\vec{X}, \vec{U}, t)$
$\vec{X} := \vec{X} + \vec{X}' \cdot dt$
end loop

Since $\mathcal{A}$ is not in general a system of equations, linear or otherwise, traditional methods do not apply. We have nonetheless been able to develop a technique for attacking such problems, which represents a generalization to algorithmic mappings of techniques appropriate to equation mappings . This approach makes use of the ***weakest precondition*** predicate transformation technique developed by Dijkstra [2] to derive solutions to questions posed of models based on algorithmic (special case: equational) mappings.

By way of example, consider once again the problem of determining steady-state conditions discussed above, this time for the case of an algorithmic mapping. For the sake of simplicity, suppose there are no exogenous variables, and time is not an explicit parameter. Thus we have

loop
$\vec{X}' := \mathcal{A}(\vec{X}, \vec{U}, t)$
$\vec{X} := \vec{X} + \vec{X}' \cdot dt$
end loop

As before, equilibrium corresponds to a condition where $\vec{X}' = \vec{0}$. In this case, however, we have an assignment statement (:=) rather than an equation (=). The question thus becomes: what has to be true *before* the assignment is performed, for $\vec{X}' = \vec{0}$ to be true *afterwards*?

This formulation is essentially identical to Dijkstra's definition of the *wp predicate transformer* [2]. For the sake of completeness, we will present a brief review of the basic predicate transformers; readers requiring more detail are referred to [2].

2.1 Introduction to the Dijkstra Transform

The *wp* predicate transformer is an operater that takes two operands, a program and a predicate, and produces a predicate as result. We will use Dijkstra's notation $wp(Prog \mid R)$ to denote the weakest (most general) predicate, called the ***weakest precondition***, that must hold prior to execution of program *Prog*, if predicate R (the ***postcondition***) is to hold after execution of *Prog*.

2.1.1 Fundamental Algorithmic Constructs

The well-known Jacopini-Böhm theorem states that any single-entry/single-exit program is equivalent to a program using only sequencing, **if** statements, and a controlled looping construct such as the **while** or **repeat-until** loop as control structures. Furthermore, loops within the main simulation loop tend to be rare, and most of the "inner syntax" of CSMs consists of assignment statements. For our purposes it therefore suffices to give the *wp* transform for sequencing, **if** statements, and assignment.

Transforms of Fundamental Algorithmic Constructs

Statement sequencing: if $S1$ and $S2$ are program statements, then $wp(S1;S2 \mid R) = wp(S1 \mid wp(S2 \mid R))$. This fact immediately generalizes to

$$wp(S1;\ldots;Sn \mid R) = wp(S1 \mid wp(S2 \mid \ldots, wp(Sn \mid R)\ldots).$$

Assignment:

$$wp(x := E \mid R) = R_{E->x}$$

where $R_{E\to x}$ denotes predicate R, with expression E substituted for all free occurrences of variable x in R. For example,

$$wp(x := 1 - y \mid x = y + 5) = y + 5 = 1 - y$$

or $y = -2$, which is the weakest predicate that had to be true if the postcondition $x = y + 5$ was to hold.

if statements The **if** statement has the usual guarded-command syntax:

$$\begin{array}{lll} \langle if \rangle ::= & \textbf{if} & \langle be_1 \rangle \rightarrow \langle statements_1 \rangle \mid \\ & \vdots & \\ & \mid & langlebe_n\rangle \rightarrow \langle statements_n \rangle \\ & \textbf{fi} & \end{array}$$

where the be_i are boolean expressions. The transformer for $\langle if \rangle$ is

$$wp(\langle if\rangle \mid R) = (\langle be_1\rangle \Rightarrow wp(\langle statements_1\rangle, R))\wedge$$
$$\vdots$$
$$(\langle be_n\rangle \Rightarrow wp(\langle statements_n\rangle \mid R))$$

In our application we can state that exactly one guard will be true. The condition $(\langle be_1\rangle \vee \ldots \vee \langle be_n\rangle)$ included in Dijkstra's formulation is therefore always true and can be omitted.

Loops The *wp* transform for looping constructs is less straightforward, and in fact cannot be stated in closed form. We assume the syntax employed in [2]:

$$\langle do\rangle ::= \textbf{do } \langle be_1\rangle \rightarrow \langle statements_1\rangle \mid$$
$$\vdots$$
$$\mid \langle be_n\rangle \rightarrow \langle statements_n\rangle$$
$$\textbf{od}$$

The semantics of this construct stipulate that the statements within the **do-od** are executed repeatedly as long as a true guard exists. As before, our application and implementation language allows us to assume that at most one guard is true for any iteration.

The transformer for $\langle do\rangle$ is

$$wp(< do >\mid R) = (\exists k \geq 0 : H_k(R))$$

where

$$H_0(R) = R \wedge \neg(\exists j : 1 \leq j \leq n : \langle be_j\rangle)$$

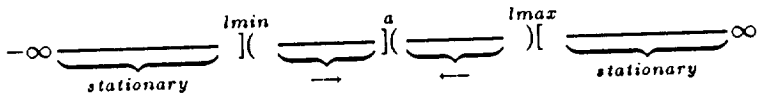
and

$$H_k(R) = wp(IF, H_{k-1}(R)) \vee H_0(R)$$

2.1.2 Applications of the Dijkstra Transform

Equilibrium Conditions

Example: An Abstract Buzzer We begin with a simple example: applying the *wp* transform to the problem of finding equilibrium conditions for an *abstract buzzer*. We assume we have a device whose (sole) moving part is characterized by its position x on the real axis. Furthermore, we posit that if $lmin < x \leq a$, then x is moving right; if $a < x < lmax$ then x is moving left; otherwise x is stationary. In schematic form:



The CSM has this form:

```
loop
{ A: }
 S1: if
       l >= lmax --> l' := 0;
       l <= lmin --> l' := 0;
       lmin < l <= a --> l' :=  1;
       a < l < lmax  --> l' := -1;
     fi
{ B: }
 S2: l := l + l' * dt
until done
```

We now pose the question: under what conditions is this system stable? In *wp* terms, this question appears as: what is the weakest precondition for $l' = 0$?

We have

$$
\begin{aligned}
l' \;=\; & \mathcal{A}(l, lmin, a, lmax) = \mathcal{A}(l) = \\
& \textbf{if} \\
& l \geq lmax \rightarrow l' := 0; \\
& l \leq lmin \rightarrow l' := 0; \\
& lmin < l \leq a \rightarrow l' := 1; \\
& a < l < lmax \rightarrow l' := -1; \\
& \textbf{fi}
\end{aligned}
$$

Then

$$
\begin{aligned}
wp(\mathcal{A}(l) \mid l' = 0) \;=\; & \\
& (l \geq lmax \Rightarrow wp(l' := 0 \mid l' = 0)) \wedge \\
& (l \leq lmin \Rightarrow wp(l' := 0 \mid l' = 0)) \wedge \\
& (lmin < l \leq a \Rightarrow wp(l' := 1 \mid l' = 0)) \wedge \\
& (a < l < lmax \Rightarrow wp(l' := -1 \mid l' = 0)) \\
& \equiv \\
& (\neg(l \geq lmax) \vee (0 = 0)) \wedge \\
& (\neg(l \leq lmin) \vee (0 = 0)) \wedge \\
& (\neg l \in (lmin, a] \vee (0 = 1)) \wedge \\
& (\neg l \in (a, lmax) \vee (0 = -1) \\
& \equiv \\
& (l \notin (lmin, a]) \wedge (l \notin (a, lmax)) \\
& \equiv
\end{aligned}
$$

$$l \notin (lmin, lmax)$$

This answer is, of course, intuitively obvious; the purpose of this example was to illustrate the formal definition of preconditions prerequisite for given post-conditions, in this case equilibrium conditions. The following example derives a less obvious precondition.

Example: Relay Servo By way of example, we will apply the wp transform to the problem of finding equilibrium conditions for the relay servo discussed previously:

$$\begin{aligned} x'' &= -x'/B + G * A/B \\ G &= \textbf{if } F(t) > x \textbf{ then } -1 \textbf{ else} \\ &\quad\ \textbf{if } F(t) > x \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \end{aligned}$$

For the sake of simplicity we first put the model in state variable form:

$$\begin{aligned} p' &= -x * A - p/B \\ q' &= -G/B \end{aligned}$$

with G as above. p represents momentum, and q is displacement in these equations.

The $\mathcal{A}$-computations of the relay servo CSM are thus

```
{S1:}  p' := -q*A - p/B
{S2:}  if F(t) < q then  q' := -1/B
       else if F(t) > q then q' := 1/B else q' := 0 fi
```

We have equilibrium if $p' = 0$ and $q' = 0$. But q' is computed by an *algorithm* (however simple), not an algebraic equation. We therefore use the "algorithm solver", the wp transform:

$$wp(S1; S2 \mid p' = 0 \wedge q' = 0) = wp(S1 \mid wp(S2 \mid p' = 0 \wedge q' = 0))$$

$$\begin{aligned} &wp(S2 \mid p' = 0 \wedge q' = 0) = \\ &\qquad (F(t) < q \Rightarrow wp(q' := -1/B \mid p' = 0 \wedge q' = 0)) \wedge \\ &\qquad (F(t) > q \Rightarrow wp(q' := 1/B \mid p' = 0 \wedge q' = 0)) \wedge \\ &\qquad (F(t) = q \Rightarrow wp(q' := 0 \mid p' = 0 \wedge q' = 0)) \\ &= \\ &\qquad (F(t) < q \Rightarrow p' = 0 \wedge -1/B = 0) \wedge \\ &\qquad (F(t) > q \Rightarrow p' = 0 \wedge 1/B = 0) \wedge \\ &\qquad (F(t) = q \Rightarrow p' = 0 \wedge 0 = 0) \end{aligned}$$

$$
\begin{aligned}
&= \quad (\textit{since } B \neq \infty, \textit{ and so } 1/B = 0 \textit{ is } \textbf{false}) \\
&\qquad (F(t) < q \Rightarrow \textbf{false}) \wedge \\
&\qquad (F(t) > q \Rightarrow \textbf{false}) \wedge \\
&\qquad (F(t) = q \Rightarrow p' = 0) \\
&= \\
&\qquad \neg(F(t) < q) \wedge \qquad \{\text{so } F(t) \geq q\} \\
&\qquad \neg(F(t) > q) \wedge \qquad \{\text{so } F(t) \leq q, \text{ i.e. } F(t) = q\} \\
&\qquad (F(t) = q \Rightarrow p' = 0) \\
&= \\
&\qquad (F(t) = q) \wedge (F(t) = q \Rightarrow p' = 0) \\
&= \\
&\qquad (F(t) = q) \wedge (p' = 0)
\end{aligned}
$$

so

$$
\begin{aligned}
&wp(S1 \mid wp(S2, p' = 0 \wedge q' = 0)) = \\
&\qquad wp(\{S1 :\}\ p' := -q * A - p/B \mid (F(t) = q) \wedge (p' = 0)) \\
&= \quad F(t) = q \wedge -q * A - p/B = 0
\end{aligned}
$$

so $F(t) = (-A/B) * p$ is the weakest precondition for equilibrium.

The preceding discussion of equilibrium conditions for a relay servo has illustrated the extension of reasoning processes heretofore possible only for algebraic mappings to algorithmic mappings by means of the *wp* transform. The mapping employed in this example was quite simple; even so, the resulting development was sufficiently complex to make it clear that more substantial examples require computer-assisted processing. The *wp* transform implementation we have developed (Appendix B) supplies the basic tools required for this sort of reasoning.

Chapter 3

Qualitative Reasoning and Interval Mathematics

We have presented an overview of CSMs, touching on dynamical systems based on equational mappings. We have given a simple canonical schematic for CSMs based on the concept of algorithmic mappings, as well as a concise vector notation for CSMs. Finally, the concept of *wp* predicate transformer was introduced and it was shown how this transform could serve as a generalized form of "solving" the algorithmic mapping, analogous to solving equational mappings, by providing answers to the question "what must hold beforehand, in order for condition P to obtain after $\mathcal{A} \cdot \mathcal{B}$ is applied?" The use of this approach was illustrated by example.

We now turn our attention to the domain of qualitative reasoning, with the ultimate goal of applying *wp* transform techniques to this domain. The rationale for employing qualitative reasoning rather than quantitative methods have been amply discussed elsewhere [3], and can be summarized as corresponding to a requirement for reasoning at a more abstract level, and/or with less specific information than is the case for quantitative reasoning. [3] contains descriptions of a number of approaches to this goal. The ontologies and methods underlying these approaches vary widely, but all have this in common: it is assumed that no quantitative model of the system of interest is available, and that inferences may/must be made solely on the basis of knowledge of monotonicity of functional relationshps, and position of values with respect to designated ("landmark") values in a quantity space.

Our research has proceeded on the basis of the opposite assumption: that a quantitative model *is* available. A number of reasons motivate this approach. Important among these is the fact that qn models *are* frequently available for the systems of interest; moreover, as pointed out in the previous discussion of spring breakage in the mass-spring model, these models often reflect physical

reality accurately enough to form a basis for ql inference about system behavior.

A second motivation for the use of qn models as inference basis is the author's belief that the most powerful model-based inference engine of all, the human mind, does its reasoning on the basis of qn rather than ql models. This view is admittedly incompatible with the assertion of most authors in the ql reasoning field that ql reasoning corresponds to the sort of modeling done by the mind. To quote Bertrand Russell, "when the experts are in agreement, the opposing position cannot be held to be certain." Nonetheless, it appears to us that the imprecision and heavy reliance on default values inherent in mental modeling has been confused with true ql reasoning processes. A simple gedankenexperiment will illustrate this point. Imagine a tennis ball dropped from shoulder height: how many times does it bounce? The gedankenexperimenter will solve this problem by producing and then watching an impromptu "mental movie" of this script. Rather than deducing or inferring a result by any formal method, the answer is obtained by counting how many times the mental image of the ball bounces on the "inner screen." Default values are inherent in the choice of production values for "shoulder height" as well as the surface on which the ball bounces. The qn nature of the process can be seen in the fact that the answer will probably "four or five times", rather than one of the answers produced by true ql reasoners: "don't know" or "infinitely often" or, most likely, an infinite branching tree of histories.

But we digress. The third cogent motive lies in these considerations:

- a qn model allows powerful inferencing, and
- if a qn model is not available, it may well be possible to produce one, even in the absence of deep physical insight. On the most basic level, this corresponds to noting which variables are present, and how they interact: if a increases, b decreases, etc. By a leap of faith, a first-cut assumption of linearity can then be made: $a = k \cdot b$. In many cases, such an assumption will not be justifiable, but may nonetheless provide a sufficiently accurate approximation of reality to be useful. [7] and [8] provide intriguing discussions of these ideas.

Why CSMs as qn Models? In view of the vast variety of qn models available, a word of explanation regarding our choice of CSMs as representation is in order. Our domain of discourse is concerned largely with *physical* systems, for which the most natural representation is in terms of relationships among magnitudes and their rates of change. If these relationships are of a particularly simple and regular form, they can conveniently be cast in the form of differential equations. If this is not the case, then the derivation of new values from old must proceed as some form of more complex computation, i.e. in terms of an algorithmic mapping. The CSM is a canonical form for such computational treatment of qn models.

Having provided a rationale for CSM-based reasoning, we now examine the processes involved, and how they differ from traditional ql reasoning. In concise terms, where ql reasoning proceeds on the basis of resolving influences, CSM-based reasoning proceeds by means of symbolic evaluation of interval values by means of interval mathematics. More precisely: as in traditional ql reasoning, there is a set of variables representing the quantities of interest in the system. Moreover, we have given a set of *landmark* values: a finite set of distinguishing points (which always include $\pm\infty$)on the extended real axis $\Re^+ = \Re \cup \{\pm\infty\}$. In our formulation each variable may have associated with it its own personal set of landmark values; that set is a totally ordered finite subset of $\Re^+$. No particular ordering need be assumed, however, for the landmarks of one variable relative to those of another.

3.1 Mathematical Background

The present section introduces the mathematical vocabulary which will be needed to develop our approach to ql simulation. Since our CSM-based reasoning system proceeds on the basis of interval arithmetic, we must first define its operands, which in Lebesque theory are known as *elementary sets* (*es*'s).

3.1.1 Definition of Elementary Sets

Definition The extended reals $\Re^+ = \Re \cup \{\pm\infty\}$, the reals augmented with $\pm\infty$.

Definition An *interval* in $\Re_p^+$ is the set of points $\mathbf{x} = (x_1, \ldots, x_n)$ such that $a_i \leq (or <)\ x_i \leq (or <)\ x_i, i = 1, \ldots, p$.

Thus intervals may be open, closed, or half open (equivalently: half closed). We will use the notation $[(a, b])$ when we wish to leave unspecified whether the endpoint in question is open or closed. [and (will be referred to as *left*,] and) as *right* marks. The *degenerate interval* $[a, a]$ is a permissible interval; it is a singleton containing only a, and will in most cases be identified with the number a. The left endpoint of the interval may be greater than the right endpoint, i.e. the empty set is an interval. In addition, if no confusion is possible, we will write $[(a, a])$ to denote the singleton *es* $\{[(a, a])\}$, and a to denote the *es* $\{[a, a]\}$.

Unless otherwise stated, we will assume that $p = 1$ i.e. our intervals are subsets of one-dimensional Euclidean space.

Definition An *elementary set* is the union of a finite number of intervals in R_p^+.

Nothing in our implementation, or in principle, requires the intervals of an *es* to be disjoint. In our application, however, *ess* do have this property in most

cases. We will use the term *overlap* to denote the intersection of two intervals within a single *es*.

It is easy to see that *es* are closed under operations such as union, intersection, and complement; in measure theoretic terms, families of sets having this properties are called *rings*. It is important to note that it is sets of intervals that are being operated on here, not the intervals in these sets.

Note that since we identify singleton intervals with the real number they contain, any finite set of real numbers is an *es* .

It will frequently be necessary to perform case analyses involving the positions of points in an *es* relative to landmarks. An operation useful in automating such analyses is the *split*, denoted by $\downarrow$. A split takes as operands an arbitrary *es* and a finite set of reals (intuitively: landmarks), and produces an *es* as result.

Definition Let E be a singleton *es* containing (only) the interval I, $I \neq \emptyset$, and let S be a singleton set containing (only) the real number a. Then $E \downarrow S$ (equivalently: $S \downarrow E$) is defined as

$$\{\{x \in I \mid x < a\}, a, \{x \in I \mid x > a\}\} \setminus \{\emptyset\}$$

As special case, we define $E \downarrow \emptyset$ to be E.

Example:

$$\begin{aligned} \{[1,2)\} \downarrow \{1.5\} &= \{[1,1.5), 1.5, (1.5,2)\} \\ \{[1,2)\} \downarrow \{0\} &= \{[1,2)\}, \textit{and} \\ \{[1,2)\} \downarrow \{1\} &= \{1,(1,2)\} \end{aligned}$$

It is easy to extend the split operator to the case where operands E and S are not singletons. If $S = \{a_1, \ldots, a_n\}$, we define $E \downarrow S$ recursively as

$$(E \downarrow \{a_1\}) \downarrow \{a_2, \ldots, a_n\}$$

If $E = \{I_1, \ldots, I_k\}$, then $E \downarrow S = (\{I_1\} \downarrow S) \cup \ldots \cup (\{I_k\} \downarrow S)$.

3.1.2 Mappings on Elementary Sets

We begin by reviewing some basic concepts of mappings. Recall that if D and R are arbitrary non-empty sets denoting the domain and range of a function $f : D \mapsto R$, and if $S \subseteq D$, then $f(S) \equiv \{f(s) \in R \mid s \in S\}$. In particular, of course, *es*s are subsets on Euclidean space, and thus functions having Euclidean space as domain extend immediately to *es*s. Furthermore, for arbitrary f, if $f : D \mapsto R$ and A and B are subsets of D, then it is easy to see that $f(A \cup B) = f(A) \cup f(B)$.

Since we are dealing with digital computers, which are necessarily finite, we must confine ourselves to finite sets. We therefore require that the mappings that occur in our application produce *es*s when applied to *es*s; we will call such

mappings *es-closed* . It is thus appropriate to discuss the issue of closure of the class of *es*s under function application.

The Mean Value Theorem implies that if I is an interval, and f is continuous, then f(I) is an interval. Thus:

Theorem Continuous function are *es*-closed.

It is easy to exhibit examples of mappings that produce non-es values when applied to *es* arguments; for example, define

$$f(x) = \begin{cases} x & \text{if } x \text{ is transcendental} \\ 0 & \text{if } x \text{ is rational} \end{cases}$$

Fortunately the sort of functions that occur as constituents of $\mathcal{A}$ and $\mathcal{B}$ mappings are better-behaved. In the case of Euler integration (as well as most other kinds), only + and * are involved in the $\mathcal{B}$ mapping; both are continuous, and thus *es*-closed.

That takes care of the $\mathcal{B}$ mapping. What of $\mathcal{A}$? The raison d'etre of CSM lies in the intractability of the $\mathcal{A}$ mapping; if $\mathcal{A}$ were well-behaved, analytic techniques would apply. Can we expect such a mapping to be *es*-closed?

Somewhat surprisingly, the answer is in the affirmative for the sorts of functions likely to occur in CSMs. These functions typically include the basic arithmetic operations such as addition, subtraction, multiplication and division, which are manifestly continuous. In fact, the higher-level operations likely to be found in CSM programs, such as trigonomentric and logarithmic functions, are invariably implemented as subprocedures composed of the four basic arithmetic operations. Thus the functions likely to occur in a CSM are all continuous.

Computer programs are composed of computational expressions that are embedded in three types of control structures: statement sequencing, **if-then-else**, and looping constructs featuring some sort of termination construct such as a **while** condition. Statement sequences that contain no control constructs compute compositions of those computational (non-branching) functions provided by the machine architecture.

It must be admitted that any modern computer provides a wide variety of additional operations, e.g. logical operations such as XOR. Such functions may not even be defined for arbitrary reals; the logical operations, in particular, operate only on integers (bit strings). We could, of course, fall back on the finite nature of the computer, and point out that a finite-state machine cannot produce an infinite set of outputs for a finite input. This argument is not satifactory, however, since it depends on the finite-grainedness of computers. Matters are clarified by postulating an abstract machine featuring unlimited memory and infinite-precision operation. In this case discontinuous operations such as boolean functions do fail to be *es*-closed, while the basic arithmetic operations retain this property. The implementation of elementary functions on computers is such that we may safely assume that such discontinuous operations do not occur in the sorts of computations that constitute the $\mathcal{A}$-computations.

It is thus apparent that the computational part (sometimes termed the *inner syntax*) of CSMs consists of compositions of continuous functions, which are in turn continuous. Discontinuity is introduced by **if** statements. Any program, however, has only a finite number of **if** statements, each of which has only a finite number of branches (usually two). The number of discontinuities introduced by **if** statements is thus finite.

Finally, we note that a **while** loop represents a finite (since we assume it terminates) number of iterations, i.e. of compositions of the mapping represented by its body, with itself. Iterating a function with at most finitely many discontinuities yields a function with at most finitely many discontinuities. We thus see that an $\mathcal{A}$ mapping is piecewise continuous, with at most finitely many discontinuities.

The above discussion has, of course, the nature of argument rather than proof. Moreover, the assurance that a mapping has only finitely many discontinuities gives no assurance that this finite number will remain bounded, or if it does, that the bound will be a representable number. As we will see subsequently, it is generally possible to reduce at least the ql *es* values of derivatives to Q_0, the space containing 0 as its only landmark. Simplification of the variables themselves is frequently possible as well.

3.2 Interval Mathematics

We now turn to the subject of *interval mathematics*, also known by the equivalent term *interval analysis*[1]. This field of mathematics, which was pioneered by R. E. Moore [6], was originally motivated by the need to formalize the study of roundoff error in computer calculations. In such an application the uncertainty regarding the value of any variable is, of course, quite small; the basic techniques nonetheless apply unchanged to the ql reasoning field, where the uncertainty is frequently on the order of "$x \in (0, \infty)$."

Interval mathematics has developed tremendously since its inception; the literature now numbers over 500 papers. Fortunately the nature of our application is such that we will require only the most basic operations in order to provide a basis for the piecewise continous functions which constitute CSMs. All such operations are implemented in terms of the machine's basic add, subtract, multiply and divide instructions, however; thus development of implementations of these basic operations was of the highest priority. We will use the term *interval arithmetic* to denote the operations of interval mathematics that implement basic arithmetic operations.

As is pointed out elsewhere in this report, when dealing with CSMs we may confine ourselves to operations relevant to continuous mathematics, and need not consider other functions provided by computer instruction sets, such

[1] Interval analysis generally concerns itself only with *closed* intervals. Our application does not allow this simplification.

as boolean functions. We will illustrate the concepts involved by showing the interval arithmetic version of addition.

The straightforward extension of any function $f : D \mapsto R$ to a function $f : 2^D \mapsto 2^R$, which has been discussed previously, gives little indication of how the result is to be constructed computationally. It is this question that interval arithmetic addresses.

3.2.1 Addition of *es* Values

Since for arbitrary function f and subsets A, B of f's domain we have $f(A \cup B) = f(A) \cup f(B)$, we may confine our discussion to singleton *es* operands, i.e. to *ess* consisting of a single interval.

Let $x = (a, b), y = (c, d)$. Then

$$\begin{aligned} x + y &= \{r \in \Re \mid \exists u, v \textit{ such that } u \in x, v \in y : r = u + v\} \\ &= (a + c, b + d) \end{aligned}$$

Similarly, if both intervals are closed, the result will be closed.

Some of the complexities that arise in these computations become apparent when we consider the case when one of the corresponding ends of the intervals is open, and the other closed, or one is $\pm\infty$, and the other is finite, or both are infinite. To consider a concrete example:

$$\begin{aligned} (-3, 3) + (2, 4) &= (-1, 7) \\ [-3, 3] + [2, 4] &= [-1, 7] \\ (-3, 3) + [2, 4] &= (-1, 7) \\ (-3, 3] + [2, 4] &= (-1, 7] \end{aligned}$$

etc.

By the $f(A \cup B) = f(A) \cup f(B)$ principle, the operations of interval arithmetic extend immediately from the case where the operands are single intervals to arbitrary *ess*. For example,

$$x + y = \{u + v \mid u \text{ an interval in x, } v \text{ an interval in } y\}$$

Interval addition is deceptively simple, possibly leading the reader to conclude that for any arithmetic operation *op*, we have $[(a, b]) \; op \; [(c, d]) = [(a \; op \; c, b \; op \; d])$. That this is not the case can be seen by considering subtraction:

$$[(a, b]) - [(c, d]) = [(a - d, b - c])$$

In particular, $[(a, b]) - [(a, b]) = [(a - b, b - a])$ rather than $[(0, 0])$.

Multiplication is defined as follows:

$$[(a, b]) * [(c, d]) = [(min(a * c, a * d, b * c, b * d), max(a * c, a * d, b * c, b * d)])$$

The reciprocal $1/[(a,b])$ of an interval $[(a,b])$ is given by $[(1/b,\ 1/a])$ if $0 \notin [(a,b])$; if $0 \in [(a,b])$ then

$$1/[(a,b]) = [-\infty, 1/a]) \cup [(1/b, \infty]$$

Finally, we have [(a,b]) / [(c,d]) = [(a,b]) * 1/[(c,d])

Since it is evident that interval arithmetic requires a computer even for simple interval addition, it was clearly necessary to implement the basic arithmetic operations +, −, *, and / for *ess*. (In fact, + and * are all that is required for the $\mathcal{B}$-computation: x := x + x' * dt). In addition, every non-trivial $\mathcal{A}$-computation will contain additional operations such as *sin*, *cos*, *exp*, etc. These were implemented as the need arose in the course of working examples. The current library of implemented *es* functions is adequate for most applications, and serves as model and basis for producing implementations of additional functions as required. For example, a tangent function for *es* can be implemented as

```
(defun es-tan (x) (es-divide (es-sin x) (es-cos x)))
```

The reader is referred to the implementation itself, reproduced in the Appendix A, for details.

3.3 Application to Qualitative Reasoning and Qualitative Simulation

The major point of the preceding development of *es* and interval mathematics concepts is that CSM-based ql simulation proceeds formally exactly as does ordinary (qn) CSM-based simulation, with the proviso that for ql simulation the operands of the CSM (more precisely: of the $\mathcal{A}$ and $\mathcal{B}$ mappings) are *ess* rather than real numbers. We begin our discussion of CSM-based ql simulation by examining the source of these *es* operands.

As is the case in traditional ql modeling, the ql CSM model contains a set of variables representing the quantities of interest in the system. Moreover, we have as given a set of *landmark* values: a finite set of distinguishing points (which always include $\pm\infty$)on the real axis $\Re^+$. In our formulation each variable has associated with it its own personal set of landmark values; that set is a totally ordered finite subset of $\Re^+$. No particular ordering need be assumed, however, for the landmarks of one variable relative to those of another.

Since the set of landmarks associated with a model variable is a set of numbers, it is clearly an *es*. We will use the notation

$$L(x) = \{l \mid l \text{ is a landmark of } x\}$$

(more precisely: $L(x) = \{[l,l] \mid l \text{ is a landmark of } x\}$.) For example, if x is the variable of the abstract buzzer example, then $L(x) = \{lmin, a, lmax\}$.

It is interesting to examine the possible values that x can assume. The value of x may be one of $lmin$, a, $lmax$, or x may be in one of the intervals $(-\infty, lmin)$, $(lmin, a)$, $(a, lmax)$, or $(lmax, \infty)$. Much more concisely, $x \in L(x) \vee x \in \overline{L(x)}$.

The preceding paragraph illustrates the fact that *ess* are a concise notation for representing qualitative variables. It is also apparent that at any time any ql variable v is either in $L(v)$ or its complement $\overline{L(v)}$. Moreover, the value of any CSM variable alternates between $L(v)$ and $\overline{L(v)}$.

It is important to emphasize at this point the fact that any variable v in any model at any time has a distinct numeric value. The *ess* thus cannot be considered to be the *value* of a model variable; rather, it represents our *knowledge* of the value of that variable. The notation $[v] = (a, b)$ does not indicate that the interval (a, b) is the value of v, but rather that the value of x is somewhere between a and b, although we are not sure exactly where. When no confusion is possible, we will become sloppy and use the notation $v = E$ (E an elementary set) rather than $[v] = E$, to denote the fact that the qualitative value of v is E, i.e. that $x \in E$.

It is evident that the traditional qualitative values are a special case of our representation, corresponding to the case $L(v) = \{0\}$. We will call this particular landmark set the Q_0 *space*. The *es* representation facilitates specifying the state of knowledge as precisely as possible. For example, if we know that v is between 1/2 and 1 or between 2 and infinity, we can indicate this by writing $[v] = \{(0.5, 1), (2, \infty)\}$, rather than the coarser-grained $x = [+]$. In addition, the *es* representation prevents the proliferation of variables occasioned by the traditional ql value space $[-], [0], [+]$. For example, to utilize the traditional space for the abstract buzzer problem it is necessary to introduce new variables:

$$\begin{aligned} x1 &= x - lmin \\ x2 &= x - a \\ x3 &= x - lmax \end{aligned}$$

Each of the new variables has only the single landmark 0; we believe, however, that such proliferation of variables radically reduces intelligibility.

ql reasoning usually deals with the ql values [−], [0], [+], and [?] (the Q_0 space).

$$\begin{aligned} [x] &= [-] \textit{ means } x \in [-\infty, 0) \\ [x] &= [0] \textit{ means } x \in [0, 0], \textit{ i.e. } x = 0 \\ [x] &= [+] \textit{ means } x \in (0, \infty] \\ [x] &= [?] \textit{ means } x \in [-\infty, \infty] \end{aligned}$$

We, however, are using a quantiative simulation as reasoning basis, and thus have a qn mapping $\mathcal{A}$ explicitly given. The Q_0 space $\{[-], [0], [+], [?]\}$ is not closed under the operations that commonly occur in CSMs. For example, if

$[x] = [+]$, then after executing $y := sin(x)$, $y \in [-1, 1]$. To map this result to the Q_0 space, we must either have $[y] = [?]$, or must case-analyze:

$$\begin{aligned} y \in [-1, 0) \equiv [y] &= [-] \\ y \in [0, 0] \equiv [y] &= [0] \\ y \in (0, 1] \equiv [y] &= [+] \end{aligned}$$

Two of these alternatives, however, discard information. Since y may well be involved in downstream computations, this is undesirable.

Chapter 4

Envisionments and Envisionment Generation

4.1 Definition and Purpose of Envisionments

The intuitive meaning of landmarks is that they are the points within the domain of each variable where "something interesting" happens. "Something interesting" is a subjective notion; in general, however, it can be stated that discontinuities are interesting, as are points where a derivative changes sign. Other points in $\Re^n$ may be interesting as well, depending on the problem at hand. The set of landmarks associated with each model variable is, after all, chosen by the user in what may be an arbitrary manner; in general, however, the points having the above-mentioned properties must be included if the ql analysis is to make sense.

Given a set of landmarks, an *operating region* of n-space is a (maximal) set of points (model variable values) such that all points in the region have the same qualitative (*es*) values, and $\mathcal{A}$-map onto the same ql derivative values.

A *qualitative state* is a set of assignments of qualitative values to the variables and derivatives of a CSM. From the above definition of operating region it is easy to see that all points of an operating region have the same qualitative state.

As a simulation proceeds, the values of model variables will, in general, transit from one operation region into another. A corresponding transition will, of course, occur from the ql state corresponding to the source region to the state representing the destination region. A state diagram depicting the ql states representing the operating regions of a ql model and the possible (subject to model constraints) transitions among these states is called an *envisionment.*

Much mainstream ql reasoning research [3] is concerned with generation and manipulation of envisionments. A significant portion of the second year of this research was devoted to a (successful) search for techniques for automatically

generating envisionments from CSMs.

Envisionments are useful for a number of reasons, most of which are thoroughly covered in [3]. As concerns the present research, ql state diagrams allow interpretation of observations of physical systems, as well as providing an abstract characterization of system behavior, correct as well as faulty. Furthermore, envisionments allow the wp transform we have developed to be applied without incurring combinatorial explosions.

4.2 Representation of Qualitative States

Since traditional ql simulation deals with ql values from Q_0 space, the corresponding ql states are accordingly vectors of values from Q_0 space:

$$x : [+]\ y : [-]\ z : [?]$$

The envisionment then consists of such states, and arrows representing possible transitions between them.

CSM-based ql simulation differs from the traditional version in that CSM envisonments contain *two* kinds of states:

Intuitively, the state of a CSM is given at any time by specifying the values of all variables occurring in the model. To maintain some semblance of tractability, we confine our attention to snapshots taken at two distinguished points in the simulation loop:

point A: the point just before the beginning of the $\mathcal{A}$-computations (equivalently: just after the end of the $\mathcal{B}$-computations)

point B: the point just after the $\mathcal{A}$-computations (equivalently: just before the beginning of the $\mathcal{B}$-computations)

A (B) states represent qualitative values of variables at loop point A (B). The significance of these points is that the states can be specified by giving the values of only the endogenous variables. At point B, the $\mathcal{A}$-computation is finished, and the exogenous and auxiliary variables that computation employed are no longer relevant. The $\mathcal{B}$-computations involve no exogenous or auxiliary variables, so once again only the endogenous variables need appear in the state specification. It is worth noting that if the CSM in question was developed by a systematic method such as bond graphs, the variables appearing in A and B states will coincide with what dynamical systems theory also terms the *state variables* of the system.

Since we are doing qualitative reasoning, the state variables will be bound to *ess*. In most cases, derivatives will employ the Q_0 space; the same may or may not be true for 0-order variables (i.e. variables which are not derivatives). If the *ess* in question are sufficiently simple, a concise graphical notation may be used. We denote A states by rectangular boxes in illustrations, or enclose them in

[...] brackets within text; since the computations for the highest derivatives are separate from the computations of all other values, we will frequently separate them by a vertical bar in the state:

$$\boxed{x_1 : es_1 \cdots x_n : es_n \mid x_1' : deriv\_es_1 \cdots x_n' : deriv\_es_n}$$

B states are represented by oval boxes having similar content, or by enclosing the state in (...). If we wish to avoid specifying the type of a state, we enclose it in [(...]).

Useful facts about CSM-envisionment states:

- Only B-states correspond to real-world situations, since only at point B of the loop are all variables at the same time point
- The values of the qualitative state variables are frequently +, -, 0, and ?, especially the variables representing derivatives

  In fact, the qualitative value of a variable may be an arbitrary finite subset of the real line.

  We usually care only about the *sign* of derivatives.
- The variables change by this pattern:

```
at B:          at A:          at B again:   at A again:

[u | v]  --> (u | v') --> [u' | v'] --> (u' | v'')
```

4.3 Generating the Envisionment from the CSM

The Starting Point Initially each member of the set of state variables has as ql value an *es*. In addition, each state variable x has associated with it a set of landmark values L(x).

4.3.1 The Set of Initial States

We begin with a definition. Recall that the configuration of a state is

$$S = [(x_1 - - - x_n \mid x_1' - - - x_n')]$$

The ql values of the derivatives are confined to [−], [0], [+], [?]; the values of the x_i, however, are arbitrary *ess*: $x_i = \{I_{i1}, \ldots, I_{in}\}$, I_{ij} an interval.

We then define the *reduction* $R = R(S)$ of S as

$$R = \{\, [(I_1 \ldots I_n \mid x'_1 \ldots x'_n])\ \mid I_j \in x_i \}$$

Thus R is the set of all states derivable from S by keeping only a single interval from each x_i of S, and making no changes to the derivatives.

The set of initial states (which are always A states, since the CSM computation begins at point A) is then the reduction

$$R(\boxed{x_1 : x_1^* \downarrow L(x_1) \cdots x_n : x_n^* \downarrow L(x_n) \mid x'_1 : [?] \cdots x'_n : [?]})$$

where x_i^* denotes the initial interval value of variable x_i. Recall that [?] is shorthand for the *es* $\{(-\infty, \infty)\}$.

For example, if the (only) state variable is x, with initial ql value $[x] = (-1, 1)$, and $L(x) = 0$, then we have $x \downarrow L(x) = \{(-1, 0), 0, (0, 1)\}$. The set of inital states is then

$$\{\, \boxed{x : (-1, 0) \mid x' : [?]}\,,\ \boxed{x : 1 \mid x' : [?]}\,,\ \boxed{x : (0, 1) \mid x' : [?]}\, \}$$

The $\mathcal{A}$-computation, of course, will assign values to x' immediately.

4.3.2 Generating the Successor B states of an Arbitrary A State

Transitions out of A States

We will begin with an example. Consider the relay servo previously discussed, which has $\mathcal{A}$-computations (state variable form)

$$\begin{aligned} p' &= -q * A - p/B \\ q' &= -G/B, \end{aligned}$$

where

$$\begin{aligned} G &= -1 \; if \; F(t) < q \\ G &= 1 \; if \; F(t) > q \\ G &= 0 \; if \; F(t) = q \end{aligned}$$

$F(t)$ is the (exogenous) input being controlled. The $\mathcal{A}$-computations of the CSM loop are thus

```
(*S1*)  p' := -q*A - p/B;
        if F(t) < q  --> q := -1/B;
           F(t) > q  --> q :=  1/B;
           F(t) = q  --> q :=  0;
        fi
```

Suppose that as before we have $A = 2, B = 0.5, V = 1$. Furthermore, suppose F(t) is constant at 0. Let the initial values of $p = (-1,0)$ and $q = (0,\infty)$. p has no landmarks; the only landmark for q occurs at $q = F(t)$, i.e. $q = 0$. We compute the set of inital states, and obtain the singleton

$$\{[p : (-1,0)\ q : (0,\infty) \mid p' : [?]\ q' : [?]]\}$$

Given these values of p and q, statement S1 taken as interval arithmetic expression produces

$$[p'] = -(0,\infty) * 2 - (-1,0)/0.5 = (-\infty,0) - (-0.5,0) = (-\infty,0)$$

At statement S2 we see that since $[q] = (0,\infty)$, only the guard $F(t) > q$ can be true. Thus $[q']$ becomes $-1/0.5 = -2$. We have thus reached the B state

$$(p : (-1,0)\ q : (0,\infty) \mid p' : (-\infty,0)\ q' : -2)$$

or, discarding superfluous derivative information,

$$(p : (-1,0)\ q : (0,\infty) \mid p' : [-]\ q' : [-])$$

Suppose that in the above example we had used $q = [0,\infty)$ as initial value instead. Then the $\downarrow$ operator would have produced

$$[p] = (-1,0),\ [q] = \{0,(0,\infty)\}$$

Applying reduction to this produces the set of (two) inital states

$$\{[p : (-1,0)\ q : [0] \mid p' : [?]\ q' : [?]], [p : (-1,0)\ q : (0,\infty) \mid p' : [?]\ q' : [?]]\}$$

Since the quard $F(t) = 0$ is true for the former state, the $\mathcal{A}$-computation now have produces B state

$$(p : (-1,0)\ q : [0] \mid p' : [-]\ q' : [0])$$

in addition to the B state computed above.

The preceding example has made it clear that **if** statements occurring within the $\mathcal{A}$-computations can lead to branching, since the destination B state depends on which guard was true. The example contains a particularly straightforward situation, insofar as the *es* values of the variables involved were such that a unique quard could be determined to be true. The general case is more complicated. Suppose we have a statement of the form

```
if {S0:} x > y then {S1:} <statement_1>
                else {S2:} <statement_2>;
```

and that $[x] = (0,2)$, $y = \{(-\infty,-1],[1,\infty)\}$. What should be the values of x and y as a result of executing this **if** statement? More precisely, it is clear that each of the two possible outcomes of the boolean expression corresponds to distinct successor values of x and y. What should they be for each of the two branches of the **if**?

The initial impulse is to require that if the S1 branch is taken, the new ql values $[x]$ and $[y]$ of x and y should be subsets of the real line such that for *arbitrary* $u \in [X], v \in [y]$ the boolean expression $x > y$ holds. Unfortunately, such a choice is not possible (more precisely: assigns $\emptyset$ to x) in this example: there is no $u \in [x]$ that guarantees that $u > v$ for arbitrary $v \in [y]$.

It is clear that we must lower our expectations, or abandon the attempt to apply $\mathcal{A}$-mappings to ql states, i.e. to *es*-valued variables. We shall proceed as follows: Suppose the state at point S0 is

$$\boxed{x : [x]\ y : [y] \mid x' : [\alpha]\ y' : [\beta]}$$

Then execution of the **if** statement produces two (more generally: one per branch) successor configurations s1 and s2. We stipulate:

$$s1 = x : \mu\ y : \nu \mid x' : \alpha\ y' : \beta$$

where

$$\mu = \{u \in [x] \mid \exists v\ in\ [y]\ such\ that\ u > v\}$$

$$\nu = \{v \in [y] \mid \exists u\ in\ [x]\ such\ that\ u > v\}$$

In words, the new ql values of x and y are sets of reals such that for any point chosen from one of the ql values, it is possible to choose a point from the ql value of the other variable such that the boolean expression is satisfied. Similarly, we have

$$s2 = x : \eta\ y : \theta \mid x' : \alpha\ y' : \beta$$

where

$$\eta = \{u \in [x] \mid \exists v\ in\ [y]\ such\ that\ u \leq v\}$$

and

$$\theta = \{v \in [y] \mid \exists u\ in\ [x]\ such\ that\ u \leq v\}$$

In terms of our example, state

$$[x : (0,2)\ y : \{(-\infty,-1],[1,\infty)\} \mid x' : [\alpha]\ y' : [\beta]]$$

produces the two sucessor configurations

$$s1 = x : (0,2)\ y : \{(-\infty,-1],[1,2)\} \mid x' : [\alpha]\ y' : [\beta]$$

and

$$s2 = x : (0,2)\ y : \{[1,\infty)\} \mid x' : [\alpha]\ y' : [\beta]$$

Let us examine the effect of this branching process more closely. Configuration s1 (s2) corresponds to the state of knowledge at program point S1 (S2). As we have seen, it is not the case that arbitrary values for x and y chosen from s1 will necessarily satisfy $x > y$. However, the meaning of the statement $[x] = E$ is that the actual value of x is unknown, but is somewhere in the es E. It is easy to see that the branching process preserves this meaning.

It is important to note that the branching process we have described does not correspond to the transition from an A state to a B state, but rather to a step *within* the $\mathcal{A}$-computation, to wit the step of executing the **if** statement in question.

It is left to the interested reader to generalize the above branching process to arbitrary sets of ql variables and arbitrary boolean expressions composed of comparisons of variables and constants connected by the usual boolean operators, as well as to verify that the resulting ql values are *ess*, i.e. **if**-branching is *es*-closed. An indication of the nature of the proof may be found by examining the algorithm that performs general **if**-branching in the implementation (Appendix A).

Here, then, is the procedure for computing the set of successor B states of a given A state s_A. Let $S_1, \ldots, S_n$ denote the statements of the $\mathcal{A}$-computation.

```
state_set := R(s_A) -- compute the reduction of s_A
for k in 1..n do
 new_state_set := emptyset;
 for_each s in stateset do
   s' := set of successors of s
         obtained upon executing S_k;
   new_state_set := new_state_set union s';
 end for_each;
 state_set := new_state_set;
end for;
-- upon termination, state_set contains
-- the set of B states that succeed s_A.
```

It is interesting to note the source of landmarks. It has been intimated that they are provided by the user as input. The above discussion, however, makes it obvious that target states are produced not only from user-supplied landmarks, but also from **if** statements. In fact, the user is not compelled to supply any landmarks at all; in the relay servo example discussed above, p had no landmarks. In such cases, all branching occurring in transitions from B to A states is produced by **if** statements. Here is a simple example: suppose we have ql variable x, for which neither landmarks nor ql value are known. We must therefore begin by assuming $[x] =?$, i.e. $x = (-\infty, \infty)$. Suppose the A computations begin with

```
x := sin(x);        -- now x = [-1,1]
```

```
if x > 0 then S1 else S2;
```

At this point *two* resulting states are created, one for $x \leq 0$ and one for $x > 0$. We thus have branching transitions, despite the lack of any information about the landmarks or value of the variable involved.

4.3.3 Generating the Successor A states of an Arbitrary B State

We continue our discussion of envisionment generation by describing the construction of transitions out of B states. The $\mathcal{B}$ mapping generally produces branching, i.e. a set of possible transitions to new (A) states. We will examine how this comes about.

At point B we have a set of values current for time t for all variables, and are about to compute subsequent values by means of equations of the form

$$x_i^{(j)}[t+1] := x_i^{(j)}[t] + x_i^{(j+1)}[t] * dt$$

where the time subscript is usually implicit. Transitions out of the present B-state are constructed in accordance with deKleer & Bobrow's principles [5], modified as required for the purposes of dealing with continuous simulation models. We repeat the original versions here for completeness:

Rule 0: Value continuity: values must change continuously over a transition, i.e. a value cannot go from - to + without assuming the value 0 at some intermediate state.

Rule 1: Contradiction avoidance: the system cannot transit to an inconsistent state. Note that this Rule fails to hold in the presence of faults.

Rule 2: Instant change rule: changes from 0 happen at an instant.
Rules 3, 4, and 5 merely state that Rules 0 and 2 also apply to derivatives; this is self-evident in our context.

Rule 6: Change to all 0 derivatives is impossible.

The following modifications and qualifications are required when applying these principles to construction of transitions out of B-states in state diagrams for continuous simulations: Rules 0 and 2 are oddities in that they do not, strictly speaking, hold for continuous simulation models: if $x[t] < 0$, then $x[t+1] = x[t] + x'[t] * dt$ can be > 0 if dt is sufficiently large. For similar reasons, a non-zero quantity may reach 0 at the same time (iteration) that another quantity becomes non-zero, thus violating Rule 2. We ignore these potential violations in qualitative reasoning, however, since they depend on the value of dt. CSM-based qualitative reasoning assumes that dt values can be made arbitrarily small, in order to provide arbitrarily close approximations to

the differential equation model. Since dt can always be chosen small enough so that the above rule violations do not occur, we posit that rules 0 and 2 hold in the qualitative domain, *for transitions out of B states.* As it happens, Rule 0 does not hold at all for transitions out of A-states.

Similar reasoning justifies Rule 6. Suppose that at point B some quantity $x_i^{(j)} <> 0$. Then $x_i^{(j-1)}$, computed by

$$x_i^{(j-1)} := x_i^{(j-1)} + x_i^{(j)} * dt$$

can equal 0 only if $x_i^{(j-1)} = -x_i^{(j)} * dt$. Again this depends on a fortuitous choice of dt, and clearly a dt can be chosen so that no variable transits to zero.

An intuitive grasp of of Rule 6 in the continuous simulation context can be obtained by noting that any simulation model can be run backwards in time by taking final values as initial values, and choosing a negative dt. Starting our backward run with all quantities equal to zero (stationary) clearly cannot result in any (simulated) motion as a result of the B computations.

Given the above considerations, we are now in a position to describe how transitions out of B states are computed. We begin by recalling that the configuration of a state is

$$S = [(x_1 - - - x_n \mid x_1' - - - x_n')]$$

The ql values of the derivatives are confined to [−], [0], [+]; the values of the x_i, however, are arbitrary *ess*: $x_i = \{I_1, \ldots, I_n\}$, I_j an interval.

Let S be an arbitrary B state. We begin the construction process by constructing a set of preliminary "scratch" configurations $R = R(S)$, the reduction of S. Given R, constructing the target A states is straightforward.

For each configuration in R:

- If any $x_i^{(j)}[t] = l$, where l is a landmark, and $x_i^{(j+1)}[t] > 0$, then transit to an A state with $x_i^{(j)}$ bound to (l, m) where m is the smallest landmark greater than l, and all other bindings unchanged (Rule 2). (Note: m may be ∞.) Similarly, if $x_i^{(j+1)}[t] < 0$, then $x_i^{(j)}$ becomes bound to (k, l), where k is the greatest landmark less than x.

  If several variables qualify, transit to a state in which each is newly bound as described above. These are the only transitions out of such a state.

- If there are no variables bound to landmark values, add arrows leading to A-state successors by identifying quantities moving to their thresholds: if $[x_i^{(j)}]$ is in the interval between two landmarks, and $[x_i^{(j+1)}] <> 0$, then $x_i^{(j)}$ is moving to a threshold, unless the end of the interval toward which it is moving is $\pm\infty$. A binding change for $[x_i^{(j)}]$ to the landmark value toward which it is moving is then possible. Add arrows for all possibilities and all combinations of possibilities of such binding changes.

Here is an example of this construction.

Let

$$S = (x : (2,3]\ y : \{-1, [0,\infty)\} \mid x' : +\ y' : -)$$

and suppose integers are landmarks.

Then

$$R = \{[(x : (2,3]\ y : -1 \mid x' : +\ y' : -)], [(x : (2,3]\ y : [0,\infty) \mid x' : +\ y' : -])\}$$

From temporary configuration $[(x : (2,3]\ y : -1 \mid x' : +\ y' : -])$ we see that x is moving right and will ultimately reach 3; y is moving left. Transitions from points happen at an instant (i.e. the time for x to reach 3 is finite, while the time it takes y to move off -1 is infinitesimal), and we see that $\boxed{x : (2,3]\ y : (-2,-1) \mid x' : +\ y' : -])}$ is one of the target A states.

Similar considerations show that the "scratch" configuration

$$[(x : (2,3] y : [0,\infty) \mid x' : +y' : -])$$

will produce the target A states

$$\begin{array}{c} [x : 3\ y : [0,\infty) \mid x' : +\ y' : -], \\ [x : (2,3]\ y : 0 \mid x' : +\ y' : -] \quad \textit{and} \\ [x : 3\ y : 0 \mid x' : +\ y' : -] \end{array}$$

The last state corresponds to x and y reaching their landmarks simultaneously.

We now discard R, and are left with the transitions

$$\begin{array}{l} (x : (2,3]\ y : \{-1, [0,\infty)\} \mid x' : +\ y' : -) \\ \quad \longrightarrow \quad [x : (2,3] y : (-2,-1) \mid x' : +y' : -] \\ \quad \longrightarrow \quad [x : 3y : [0,\infty) \mid x' : +y' : -], \\ \quad \longrightarrow \quad [x : (2,3] y : 0 \mid x' : +y' : -] \quad \textit{and} \\ \quad \longrightarrow \quad [x : 3y : 0 \mid x' : +y' : -] \end{array}$$

It is clear that the tedious and painstaking nature of this process makes the computer implementation indispensable.

4.4 Examples

4.4.1 An Abstract Buzzer

A program was written to generate envisionments automatically. Figure 4.1 shows the envisionment produced for the abstract buzzer: It should be pointed out that the graphic state diagram was created by hand from the actual output of the envisionment generator, which produces a list of states and transitions.

```
loop
  (A)
S1: if
      l >= lmax --> l' := 0;
      l <= lmin --> l' := 0;
      lmin < l & l <= a --> l' := 1;
      a < l & l < lmax --> l' := -1;
    fi
  (B)
S2: l := l + l' * dt
  until done;
```

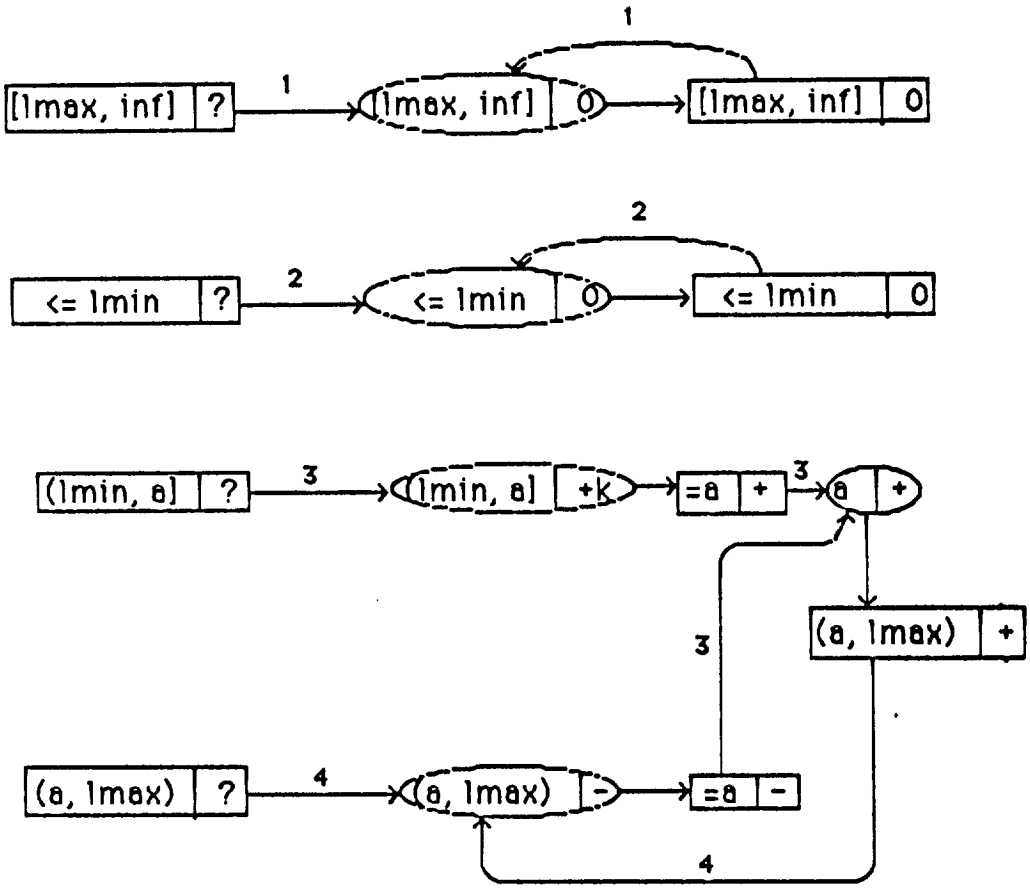


Figure 4.1: Envisionment of Abstract Buzzer

4.5 An AEROBEE Rocket Control System

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 [1] for further discussion. In outline form, the system operates as follows. The free gyro produces an error voltage $EV = G2 * sin(RA - X)$, where $G2$ is the gyro sensitivity, RA the reference angle (i.e. the intended angle of orientation), and X is the actual angle of orientation of the rocket. This error voltage EV is added with a feedback voltage $FV = -G1 * X'$ 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 the network is given by $V = C1 * EV + C2 * FV$, where

$$C1 = 1/(1 + (1 + R3/R2) * (R1/R3)), and$$
$$C2 = 1/(1 + (1 + R3/R1) * (R2/R3))$$

The dynamics of the rocket can be taken as $X'' = F * MA/I$, where MA 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 thrustors used to control the angular orientation. These thrustors can produce only three discrete levels of forces: FA, 0, and −FA; there is a dead space of $2 * DS$. 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 '(g1 := 0.13)) ; rate gyro sensitivity
(setq C6 '(g2 := 11.9)) ; free gyro sensitivity
(setq C7 '(ds := 0.025)) ; dead space in thrustor
(setq C8 '(r1 := 33000)) ; resistance R1 in mixing net
(setq C9 '(r2 := 33000)) ; resistance R2 in mixing net
(setq C10 '(r3 := 25000)) ; resistance R3 in mixing net
(setq C11 '(pi := 3.14159))
(setq C12 '(x := 12.08))     ; orientation of rocket
(setq C13 '(x1dot := -1.27)) ; 1st 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)*(r1/r3)) ))
  ; mixing network
(setq M2  '(c2 = 1/(1+(1+r3/r1)*(r2/r3))  ))
  ; coefficients

; statements
(setq S1 '(ev := g2 * sin((ra - x) * rc) ))
  ; error voltage
(setq S2 '(fv := NEG g1 * x1dot))   ; feedback voltage
(setq S3 '(v := c1 * ev + c2 * fv)) ; network output voltage

(setq S4 '(IF
              ((v < NEG ds) ==> (h := v + ds))
              ((v <= ds)    ==> (h := 0)     )
              ((ds < v)     ==> (h := v - ds))
)          )   ; thrustor controller dead space

(setq S5 '(IF
              ((h < 0) ==> (f := -4))
              ((h = 0) ==> (f :=  0))
              ((0 < h) ==> (h :=  4))
)          )   ; thrustor force

(setq S6 '(x2dot := (f * ma / i) * dc))
  ; rocket dynamics

; part B of the simulation: updating of x and x'
(setq S7 '(x := x + x1dot * dt))
(setq S8 '(x1dot := x1dot + x2dot * dt))
```

The above program statements constitute parts A and B of the simulation loop for the Aerobee attitude controller.

Here is the envisionment produced for the aerobee rocket:

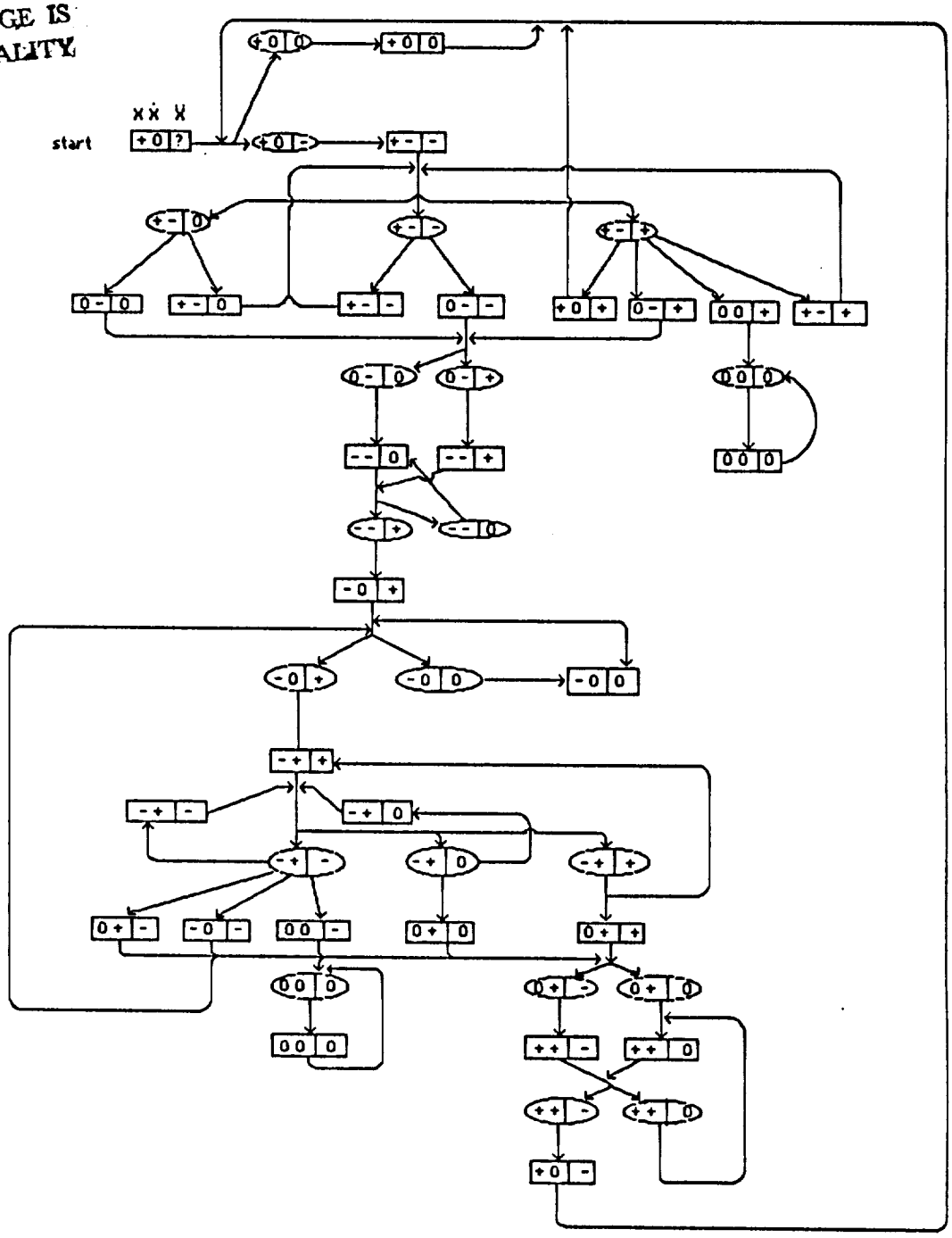


Figure 4.2: Envisionment of Aerobee Rocket

Chapter 5

CSM-based Fault Diagnosis

5.1 Use of Envisionments for wp-based Diagnosis

A major advantage of having available an envisionment of the system being simulated is that it is frequently possible to use wp transform techniques on envisionments, whereas the same approach would lead to a combinatorial explosion if applied to the raw output data.

The application of the wp transform to envisionments depends on the assumption that the structure of the envisionment remains valid despite the occurrence of faults. This corresponds to the assumption, discussed previously, that faults correspond to changes in model parameters but leave model structure unchanged.

5.1.1 Examples of Use

Buzzer Diagnosis using wp

Our first example of wp-based diagnosis will consider a particularly simple system: the abstract buzzer introduced previously. We reproduce the CSM here:

```
loop
{ A: }
 S1: if
       l >= lmax --> l' := 0;
       l <= lmin --> l' := 0;
       lmin < l <= a --> l' :=  1;
       a < l < lmax  --> l' := -1;
     fi
```

```
{ B: }
 S2: l := l + l' * dt
until done
```

As indicated, we will assume that the model structure remains valid in the presence of faults, i.e. that there are numbers $lmin$, $lmax$, and a, such that if l is in the stated relationship to these points, then it *will* move with speed 0 or ± 1, depending. Faults are assumed to correspond to inadvertent changes to $lmin$, $lmax$, and a.

Let $lmin$ and $lmax$ have nominal values -2 and 2, respectively. Now suppose that we observe the following phenomenon *in the real system*: l is stationary at -3, but suddenly begins moving to the right at velocity 1. This corresponds to a state transition

$$1 : (l : (-\infty, lmin] \mid l' : 0) \longrightarrow 2 : (l : \ldots \mid l' : 1)$$

Both states 1 and 2 must be B states, since only B states correspond to states of the real-world system. A search of the set of B states for the buzzer system retrieves only two B states whose ql value for l' is [+]. These states are $2 : (l : (lmin, a] \mid l' : [+])$ and $2' : (l : a \mid l' : [+])$

We now ask: what is the weakest precondition that l' be [+] after transiting from state 1 to an unknown intermediate A state, and thence to state 2? In terms of the computations, this is equivalent to the query "what is the weakest precondition that l' be [+] after executing statement S2, then S1?"

This formulation omits the critical fact that l was observed to be stationary at -3 before starting to move right. We thus need to append "given that $l = -3$ and $l' = 0$ before executing S2; S1." Formally, we thus have

```
wp({V1:} l := -3; {V2:} l' := 0; S2; S1 | l' = 1)
```

The statements labeled V1 and V2 express the "initially stationary at -3" condition. Executing these assigments before S2; S1 assures that the desired values are bound to l and l' before the transitions are taken.

Here are the wp calculations for this system:

$$\begin{aligned}
& wp(V1; V2; S2; S1 \mid l' = 1) \\
&\quad = \quad wp(V1; V2; S2 \mid wp(S1 \mid l' = 1)) \\
&\quad = \\
&\qquad wp(V1; V2; S2 \mid (l \geq lmax \Rightarrow wp(l' := 0 \mid l' = 1)) \wedge \\
&\qquad (l \leq lmin \Rightarrow wp(l' := 0 \mid l' = 1)) \wedge \\
&\qquad (lmin < l \leq a \Rightarrow wp(l' := 1 \mid l' = 1)) \wedge \\
&\qquad (a < l < lmax \Rightarrow wp(l' := -1 \mid l' = 1))) \\
&\quad = \\
\end{aligned}$$

$$wp(V1; V2; S2 \mid (l \geq lmax \Rightarrow 1 = 0) \wedge$$
$$(l \leq lmin \Rightarrow 1 = 0) \wedge$$
$$(lmin < l \leq a \Rightarrow 1 = 1) \wedge$$
$$(a < l < lmax \Rightarrow 1 = -1))$$

Simplifying

$$(l \geq lmax \Rightarrow 1 = 0) \wedge$$
$$(l \leq lmin \Rightarrow 1 = 0) \wedge$$
$$(lmin < l \leq a \Rightarrow 1 = 1) \wedge$$
$$(a < l < lmax \Rightarrow 1 = -1)$$

yields

$$(l < lmax \vee \mathbf{false}) \wedge \tag{5.1}$$
$$(l > lmin \vee \mathbf{false}) \wedge \tag{5.2}$$
$$(lmin < l \leq a \vee 1 = 1) \wedge \tag{5.3}$$
$$(l \notin (a, lmax) \vee \mathbf{false}) \tag{5.4}$$

which is equivalent to

$$(l < lmax) \wedge (l > lmin) \wedge (lnot \in (a, lmax))$$

i.e. $l \in (lmin, a]$. The original expression

$$wp(V1; V2; S2; S1 \mid l' = 1)$$

thus reduces to

$$wp(V1; V2; S2 \mid l \in (lmin, a])$$

Since S2 is $l := l + l'dt$, this becomes

$$wp(\{V1 :\}\ l := -3; \{V2 :\}\ l' := 0; \mid l + l'dt \in (lmin, a]) =$$
$$wp(\{V1 :\} l := -3; \mid l \in (lmin, a])$$
$$\equiv \quad -3 \in (lmin, a]$$

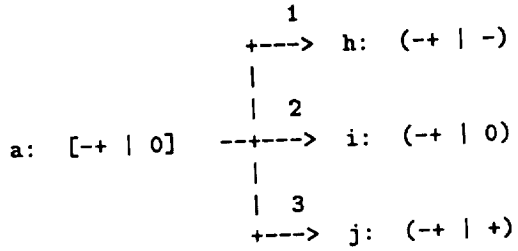
i.e.

$$(lmin < -3) \wedge (-3 \leq a)$$

The putative values of *lmin* and *a* are -2 and 0 respectively. $-3 < 0 = a$ is **true**, so we have $(lmin < -3) \wedge \mathbf{true}$, or eqivalently, $(lmin < -3)$ as the weakest precondition for the observed symptoms: lmin has shifted.

5.2 Relay Servo Diagnosis using wp

Our second example shows a wp-based diagnosis performed on a more complex system: the relay servo. The ql state diagram for the relay servo contains the following transitions:



(The numbers on the arrows, like the labels on the states, are for reference purposes only.) None of the states h, i, or j violates any constraints, and thus these transitions must be included in the state diagram. But upon observing the actual relay servo in action, we notice that it is always transition 3 that is in fact taken; it then appears at least plausible that transitions 1 and 2 represent abnormal conditions. Now suppose that one day we find the relay servo making unusual noises, and find that the controlled variable X is oscillating strangely. Upon making some numeric measurements to determine X and X' values[1] from the *actual* servo (*not* the simulation model), we find that the actual system is taking transition 1. What does this signify? As was the case for the abstract buzzer, we apply the wp approach to find out. Suppose the values $X = -0.2$, $X' = 0.8$ have been observed. (These values were taken from [1], Fig. 3.60, p. 120.) We now pose the question "what has to hold in order for transition 1 to be taken, given that $X = -0.2$, $X' = 0.8$?" Recall that the transition in question corresponds to proceeding from point A to point B, and so the problem is formulated as

$$wp(PROG \mid X'' < 0)$$

where PROG denotes the program segment

```
-- statement 1:
if X < 0 then G := 1;
elsif X = 0 then G := 0;
else G := -1;
end if;
-- statement 2:
X'' := -X'/B + G * A/B
```

[1] X'' can, if necessary, be observed as well.

In other words, what is the most general statement that must hold before executing PROG, if $[X'']$ = − is to be true afterwards? Proceeding as usual, we find

$$wp(statement\ 2 \mid X'' < 0) \equiv -X'/B + G * A/B < 0$$

and so $G * A/B < X'/B$. Since by physical considerations $B > 0$, we simplify this to $G * A < X'$, and since $X' = 0.8$, we have $G * A < 0.8$. Continuing:

$$\begin{aligned} wp(statement\ 1 \mid & G * A < 0.8) = \\ & X < 0 \Rightarrow wp(G := 1 \mid G * A < 0.8); \\ & X = 0 \Rightarrow wp(G := 0 \mid G * A < 0.8); \\ & \textbf{true} \Rightarrow wp(G := -1 \mid G * A < 0.8); \\ & \textbf{end if}; \end{aligned}$$

Since $X = -0.2 < 0$, we have $wp(G := 1 \mid G * A < 0.8)$, which is $1 * A < 0.8$. But this contradicts the *supposed* binding of A to 2, and so we have a malfunction consisting of a shift in the value of A: the loop gain has inadvertently decreased.

5.3 Future Research Directions

The techniques we have described were applied to a number of additional systems. In particular, experiments in envisionment-based diagnosis were carried out on the automatically generated state diagram of an aircraft carrier arresting cable system described in [1]. This system was used to explore hypotheses such as the possibility that the *track* of the actual system, i.e the sequence of states it traced out in the envisionment, could yield valuable diagnostic clues. Initial investigations proved promising, but the research project ended before definitive conclusions could be reached. It was apparent, however, that additional research was needed on problems such as appropriate notations for the characterization of tracks, as well as the integration of quantitative information with the envisionment.

Bibliography

[1] Y. Chu, *Digital Simulation of Continuous Systems*, Mcgraw-Hill, New York, 1969.

[2] E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[3] D. Bobrow, ed., *Qualitative Reasoning about Physical Systems*, MIT, Press, Cambridge, MA, 1985.

[4] R. Rosenberg & D. Karnopp, *System Dynamics: a Unified Approach*, Wiley & Sons, New York, 1975.

[5] J. de Kleer & D. Bobrow, *Qualitative Reasoning with Higher-Order Derivatives*, Proc. of the 1984 National Conference on Artificial Intelligence, Austin, TX.

[6] R. Moore, *Methods and Applications of Interval Analysis*, SIAM, Philadephia, 1979.

[7] C. Puccia & R. Levis, *Qualitative Modeling of Complex Systems*, Harvard University Press, Cambridge, MA, 1985.

[8] F. Roberts, *Discrete Mathematical Models*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

Appendix A

Envisionment Generator

```
;                              states.part1

{Used with states.part2 and a specialized X.part3 to construct
state space transitions for the simulation in X.part3,
according to the scheme outlined in the proposal.}

{TO RUN:
  Compile states.part1, states.part2 and X.part3 and type
                              (main_loop).
  Go for coffe AND dinner}

;************************ DATA STRUCTURES ************************
;
;ENDPOINT -- a list containing the value of an endpoint and either
;            'c' if the endpoint is included in the range or 'o'
;            if it is not.
;                 EX: (14 o)
;INTERVAL -- EITHER a single atom, indicating left and right
;            endpoints are the same (i.e., a single known value)
;                 EX: 7
;                 EX: infinity
;            OR a list consisting of a left and a right ENDPOINT
;                 EX: ((14 o)(25 c))
;VALUE --    list of INTERVALs
;                 EX: ( 7 ((14 o)(25 c)) ((-34 c)(0 c)) infinity)
;            The example contains 4 intervals.
;
;*****************************************************************
```

```
; ** Returns value of left endpoint of an interval
; ** RANGE is an interval
(defun lefft (range)
  (if (atom range) range (caar range))
  )

; ** Returns value of right endpoint of an interval
; ** RANGE is an interval
(defun rite (range)
  (if (atom range) range (caadr range))
  )

; ** Returns 'c' if RANGE is a single value, ot otherwise the
; ** 'o' or 'c' indication from the left endpoint.
; ** RANGE is an interval
(defun loc (range)
  (if (atom range) 'c (cadar range))
  )

; ** Returns 'c' if RANGE is a single value, ot otherwise the
; ** 'o' or 'c' indication from the right endpoint.
; ** RANGE is an interval
(defun roc (range)
  (if (atom range) 'c (cadadr range))
  )

; ** Returns t if the left side of the interval is open;
; ** Otherwise nil.
; ** RANGE is an interval.
(defun openleft (range)
  (equal (loc range) 'o)
  )

; ** Returns t if the left side of the interval is closed;
; ** Otherwise nil.
; ** RANGE is an interval.
(defun closedleft (range)
  (equal (loc range) 'c)
  )

; ** Returns t if the right side of the interval is open;
; ** Otherwise nil.
; ** RANGE is an interval.
(defun openright (range)
```

```
  (equal (roc range) 'o)
  )

; ** Returns t if the right side of the interval is closed;
; ** Otherwise nil.
; ** RANGE is an interval.
(defun closedright (range)
  (equal (roc range) 'c)
  )

; ** If left endpoint of RANGE is greater than right endpoint, then
; ** the left and right endpoints are swapped. Otherwise RANGE is
; ** returned unchanged.
; ** RANGE is an interval.
(defun order (range)
  (cond
    ((atom range) range)
    ((or (equal (lefft range) minf) (equal (rite range) inf)) range)
    ((or (equal (lefft range) inf) (equal (rite range) minf))
     (reverse range))
    ((<= (lefft range) (rite range)) range)
    (t (reverse range))
    )
  )

; ** Returns t if any point in interval A is less than any point
; ** in interval B.
; ** A and B are intervals.
(defun one_less (a b)
  (cond
    ((or (equal (lefft a) minf) (equal (rite b) inf)) t)
    ((or (equal (lefft a) inf) (equal (rite b) minf)) ())
    (t (< (lefft a) (rite b)))
    )
  )

; ** Returns an interval that is the portion of interval A that is
; ** strictly less than some point in interval B.
; A and B are intervals.
(defun prune_less (a b)
  (cond
    ((or (equal (rite b) inf)
         (equal (rite a) minf)) a)
    ((or (equal (rite a) inf)
```

```
        (equal (rite b) minf)
        (>= (rite a) (rite b)))
    (list (list (lefft a) (loc a)) (list (rite b) 'o)))
   (t a)
   )
 )

; ** Returns an interval that is the portion of interval A that is
; ** strictly greater than some point in interval B.
; A and B are intervals.
(defun prune_greater (a b)
  (cond
    ((or (equal (lefft b) minf)
         (equal (lefft a) inf)) a)
    ((or (equal (lefft a) minf)
         (equal (lefft b) inf)
         (<= (lefft a) (lefft b)))
     (list (list (lefft b) 'o) (list (rite a) (roc a))))
    (t a)
    )
  )

; ** Returns true if any interval in A contains a point which is
; ** less than a point in any interval in B.
; ** A and B are VALUES (lists of intervals)
(defun qual_less (a b &aux aux1)
  (dotimes (k (length a))
           (dotimes (i (length b))
                    (if (one_less (nth k a) (nth i b))
                        (setq aux1 t)
                        )
                    )
           )
  aux1
  )

; ** Returns a value that contains intervals of A that are less than
; ** some point in an interval of B.
; ** A and B are values.
(defun find_less (a b &aux aux1)
  (dotimes (k (length a))
           (dotimes (i (length b))
                    (if (one_less (nth k a) (nth i b))
                        (setq aux1 (cons (prune_less (nth k a) (nth
```

```
i b)) aux1))
                        )
                    )
            )
  aux1
  )

; ** Returns a value that contains intervals of A that are greater
; ** than some point in an interval of B.
; ** A and B are values.
(defun find_greater (a b &aux aux1)
  (dotimes (k (length a))
           (dotimes (i (length b))
                    (if (one_greater (nth k a) (nth i b))
                        (setq aux1 (cons (prune_greater (nth k a) (nth
i b)) aux1))
                        )
                    )
           )
  aux1
  )

; ** Returns t if interval A and interval B are disjoint.
; ** A and B are intervals.
(defun one_ne (a b)
  (or
    (one_less (rite a) (lefft b))
    (one_less (rite b) (lefft a))
    (and (equal (rite a) (lefft b))
         (or (openright a) (openleft b)))
    (and (equal (lefft a) (rite b))
         (or (openleft a) (openright b)))
    )
  )

; ** Returns t if any point in interval A is equal to any point
; ** in interval B.
; ** A and B are intervals.
(defun one_eq (a b)
  (null (one_ne a b))
  )


; ** Returns true if any interval in A contains a point which is
```

```
; ** equal to a point in any interval of B.
; ** A and B are VALUES (lists of intervals).
(defun qual_eq (a b &aux aux1)
  (dotimes (k (length a))
           (dotimes (i (length b))
                    (if (one_eq (nth k a) (nth i b))
                        (setq aux1 t)
                        )
                    )
           )
  aux1
  )

; ** Returns true if all intervals of A are disjoint from all
; ** intervals of B.
; ** A and B are VALUES (lists of intervals).
(defun qual_ne (a b)
  (null (qual_eq a b))
  )

; ** Returns true if any interval in A contains a point which is
; ** less than or equal to a point in any interval in B.
; ** A and B are VALUES (lists of intervals)
(defun qual_lesseq (a b)
  (or (qual_less a b) (qual_eq a b))
  )

; ** Returns t if any point in interval A is greater than any point
; ** in interval B.
; ** A and B are intervals.
(defun one_greater (a b)
  (cond
    ((or (equal (lefft a) inf) (equal (rite b) minf)
         (equal (rite a) inf) (equal (lefft b) minf)) t)
    ((or  (equal (rite a) minf) (equal (lefft b) inf)) ())
    (t (> (rite a) (lefft b)))
    )
  )

; ** Returns true if any interval in A contains a point which is
; ** greater than a point in any interval in B.
; ** A and B are VALUES (lists of intervals).
(defun qual_greater (a b &aux aux1)
  (dotimes (k (length a))
```

```
        (dotimes (i (length b))
                 (if (one_greater (nth k a) (nth i b))
                     (setq aux1 t)
                     )
                 )
        )
 aux1
 )


; ** Returns true if any interval in A contains a point which is
; ** greater than or equal to a point in any interval in B.
; ** A and B are VALUES (lists of intervals)
(defun qual_greatereq (a b)
  (or (qual_greater a b) (qual_eq a b))
  )

; ** Conses A onto B only if B doesn't already contain A.
; ** B is a list -- A is anything you want it to be.
(defun mycons (a b)
  (cond
    ((member a b) b)
    (t (cons a b))
    )
  )

; ** Returns the greater of two endpoints.
; ** A and B are endpoints. They may not be constants.
(defun onemax (a b)
  (cond
    ((or (equal (car b) minf) (equal (car a) inf)) a)
    ((or (equal (car a) minf) (equal (car b) inf)) b)
    ((> (car a) (car b)) a)
    ((> (car b) (car a)) b)
    ((equal (cadr a) 'o) b)
    (t a)
    )
  )

; ** Returns the maximum valued endpoint from the list of endpoints
; ** A, or the endpoint B.
; ** A is a list of endpoints and B is an endpoint.
; ** The endpoints may not be constants.
(defun mymaxaux (a b)
```

```
  (cond
    ((null a) b)
    (t (mymaxaux (cdr a) (onemax (car a) b)))
    )
  )

; ** Returns the maximum valued endpoint from the list of endpoints
; ** A. The endpoints may not be constants.
(defun mymax (a)
  (mymaxaux a (list minf 'c))
  )

; ** Returns the lesser of two endpoints.
; ** A and B are endpoints. They may not be constants.
(defun onemin (a b)
  (cond
    ((or (equal (car b) minf) (equal (car a) inf)) b)
    ((or (equal (car a) minf) (equal (car b) inf)) a)
    ((> (car a) (car b)) b)
    ((> (car b) (car a)) a)
    ((equal (cadr a) 'o) b)
    (t a)
    )
  )

; ** Returns the minimum valued endpoint from the list of endpoints
; ** A, or the endpoint B.
; ** A is a list of endpoints and B is an endpoint.
; ** The endpoints may not be constants.
(defun myminaux (a b)
  (cond
    ((null a) b)
    (t (myminaux (cdr a) (onemin (car a) b)))
    )
  )

; ** Returns the minimum valued endpoint from the list of endpoints
; ** A. The endpoints may not be constants.
(defun mymin (a)
  (myminaux a (list inf 'c))
  )

; ** Returns the value portion of an endpoint A, which may or may
; ** not be a constant.
```

```
(defun int_val (a)
  (if (atom a) a (car a))
  )

; ** Returns 'c if a is a constant, or the 'o' or 'c' portion of
; ** the interval A otherwise.
(defun intoc (a)
  (if (atom a) 'c (cadr a))
  )

; ** Returns the sum of two values A and B, either numeric or
; ** symbolic (infinity or negative_infinity).
(defun plusval (a b)
  (cond
    ((or (equal a inf) (equal b inf)) inf)
    ((or (equal a minf) (equal b minf)) minf)
    (t (+ a b))
    )
  )

; ** Returns 'o if either a or b is 'o. Otherwise 'c.
(defun plusoc (a b)
  (if (or (equal a 'o) (equal b 'o)) 'o 'c)
  )

; ** Returns the sum of two endpoints A and B, either numeric or
; ** symbolic (infinity or negative_infinity).
(defun plusint (a b)
  (list (plusval (int_val a) (int_val b)) (plusoc (intoc a) (intoc
b)))
  )

; ** Returns the left endpoint of the interval A.
(defun leftpt (a)
  (if (atom a) a (car a))
  )

; ** Returns the right endpoint of the interval A.
(defun rightpt (a)
  (if (atom a) a (cadr a))
  )

; ** Returns an ordered interval representing the sum of the two
; ** intervals A and B.
```

```
(defun one_plus (a b)
  (order (list (plusint (leftpt a) (leftpt b))
               (plusint (rightpt a) (rightpt b))))
  )

; ** Returns a VALUE that contains intervals that result from
; ** all pairwise sums of intervals, one from A and one from B.
; ** A and B are VALUES.
(defun qual_plus (a b &aux aux1)
  (dotimes (k (length a))
           (dotimes (i (length b))
                    (setq aux1
                          (cons (one_plus (nth k a) (nth i b)) aux1))
                    )
           )
  aux1
  )


; ** Returns the difference of two values A and B, either numeric
; ** or symbolic (infinity or negative_infinity).
(defun minusval (a b)
  (cond
    ((or (equal a inf) (equal b minf)) inf)
    ((or (equal a minf) (equal b inf)) minf)
    (t (- a b))
    )
  )

; ** Returns 'o if either a or b is 'o. Otherwise 'c.
(defun minusoc (a b)
  (if (or (equal a 'o) (equal b 'o)) 'o 'c)
  )

; ** Returns the difference of two endpoints A and B, either numeric
; ** or symbolic (infinity or negative_infinity).
(defun minusint (a b)
  (list (minusval (int_val a) (int_val b)) (minusoc (intoc a) (intoc
b)))
  )

; ** Returns an ordered interval representing the difference of the
; ** two intervals A and B.
(defun one_minus (a b)
```

```
  (order (list (minusint (leftpt a) (rightpt b))
               (minusint (rightpt a) (leftpt b))))
  )

; ** Returns a VALUE that contains intervals that result from
; ** all pairwise differences of intervals, one from A and one from
; ** B.
; ** A and B are VALUES.
(defun qual_minus (a b &aux aux1)
  (dotimes (k (length a))
           (dotimes (i (length b))
                    (setq aux1
                          (cons (one_minus (nth k a) (nth i b)) aux1))
                    )
           )
  aux1
  )

; ** Returns a list consisting of the minimum and maximum
; ** endpoints from the list of endpoints X.
(defun getmaxandmin (x)
  (list (mymin x) (mymax x))
  )

; ** Returns the product of two values A and B, either numeric or
; ** symbolic (infinity or negative_infinity).
(defun timesval (a b)
  (cond
    ((or (equal a 0) (equal b 0)) 0)
    ((and (equal a inf) (equal b inf)) inf)
    ((and (equal a inf) (equal b minf)) minf)
    ((and (equal a inf) (< b 0)) minf)
    ((equal a inf) inf)
    ((and (equal a minf) (equal b minf)) inf)
    ((and (equal a minf) (equal b inf)) minf)
    ((and (equal a minf) (< b 0)) inf)
    ((equal a minf) minf)
    ((and (equal b inf) (> a 0)) inf)
    ((equal b inf) minf)
    ((and (equal b minf) (< a 0)) inf)
    ((equal b minf) minf)
    (t (* a b))
    )
  )
```

```
; ** Returns 'o if either a or b is 'o. Otherwise 'c.
(defun timesoc (a b)
  (if (or (equal a 'o) (equal b 'o)) 'o 'c)
  )

; ** Returns the product of two endpoints A and B, either numeric or
; ** symbolic (infinity or negative_infinity).
(defun timesint (a b)
  (list (timesval (int_val a) (int_val b)) (timesoc (intoc a) (intoc
b)))
  )

; ** Returns an ordered interval representing the product of the two
; ** intervals A and B.
(defun one_times (a b)
  (getmaxandmin (list (timesint (leftpt a) (leftpt b))
                      (timesint (rightpt a) (rightpt b))
                      (timesint (leftpt a) (rightpt b))
                      (timesint (rightpt a) (leftpt b))
                      )
                )
  )


; ** Returns a VALUE that contains intervals that result from
; ** all pairwise products of intervals, one from A and one from B.
; ** A and B are VALUES.
(defun qual_times (a b &aux aux1)
  (dotimes (k (length a))
           (dotimes (i (length b))
                    (setq aux1
                          (cons (one_times (nth k a) (nth i b)) aux1))
                    )
           )
  aux1
  )

; ** Returns the quotient of two values A and B, either numeric or
; ** symbolic (infinity or negative_infinity).
(defun divideval (a b)
  (cond
    ((equal a 0) 0)
    ((and (equal a inf) (equal b inf)) inf)
```

```
   ((and (equal a inf) (equal b minf)) minf)
   ((and (equal a inf) (< b 0)) minf)
   ((equal a inf) inf)
   ((and (equal a minf) (equal b minf)) inf)
   ((and (equal a minf) (equal b inf)) minf)
   ((and (equal a minf) (< b 0)) inf)
   ((equal a minf) minf)
   ((equal b inf) 0)
   ((equal b minf) 0)
   (t (/ a b))
   )
  )

; ** Returns 'o if either a or b is 'o. Otherwise 'c.
(defun divideoc (a b)
  (if (or (equal a 'o) (equal b 'o)) 'o 'c)
  )

; ** Returns the quotient of two endpoints A and B, either numeric
; ** or symbolic (infinity or negative_infinity).
(defun divideint (a b)
  (list (divideval (int_val a) (int_val b)) (divideoc (intoc a) (intoc
b)))
  )

; ** Returns an ordered interval representing the quotient of the
; ** two intervals A and B.
(defun one_divide (a b)
  (getmaxandmin (list (divideint (leftpt a) (leftpt b))
                      (divideint (rightpt a) (rightpt b))
                      (divideint (leftpt a) (rightpt b))
                      (divideint (rightpt a) (leftpt b))
                      )
                )
  )


; ** Returns a VALUE that contains intervals that result from
; ** all pairwise quotientss of intervals, one from A and one from
; ** B.
; ** A and B are VALUES.
(defun qual_divide (a b &aux aux1)
  (dotimes (k (length a))
           (dotimes (i (length b))
```

```
                    (setq aux1
                          (cons (one_divide (nth k a) (nth i b)) aux1))
                    )
            )
  aux1
  )

; ** Returns the sine of the value A, either numeric or symbolic
; ** (infinity or negative_infinity). An interval is returned.
(defun one_sin (a &aux aux1)
  (if (one_less a 0) (setq aux1 '(((-1 c) (0 o)))))
  (if (one_eq a 0) (setq aux1 (cons 0 aux1)))
  (if (one_greater a 0) (setq aux1 (cons '((0 o) (1 c)) aux1)))
  aux1
  )

; ** Returns a list of intervals, one for the sine of each
; ** interval in the list of intervals A.
(defun qual_sin (a  &aux aux1)
  (dotimes (k (length a))
           (setq aux1
                 (append (one_sin (nth k a)) aux1))
           )
  aux1
  )


;sub performs a textual substitution of x for all occurrences of
;z in y (at any level of nesting).
(defun sub (x in y for z)
  (cond
    ((null y) ())
    ((null (atom (car y)))
     (cons (sub x in (car y) for z) (sub x in (cdr y) for z))
     )
    ((eq (car y) z)
     (cons x (sub x in (cdr y) for z))
     )
    (t (cons (car y) (sub x in (cdr y) for z)))
    )
  )
```

```
;* dead space function
(defun dsp (a b c &aux aux1)
  (if (qual_less a b)
      (setq aux1 (qual_minus (find_less a b)
                             (find_greater b (find_less a b)))))
  (if (and (qual_lesseq b a) (qual_lesseq a c))
      (setq aux1 (cons 0 aux1)))
  (if (qual_greater a c)
      (setq aux1 (append (qual_minus
                          (find_greater a c)
                          (find_less c (find_greater a c))) aux1)))
  aux1
  )

;* function switch
(defun fsw (a b c d &aux aux1)
  (if (qual_less a '(0)) (setq aux1 b))
  (if (qual_eq a '(0)) (setq aux1 (append c aux1)))
  (if (qual_greater a '(0)) (setq aux1 (append d aux1)))
  aux1
  )

(defun msetq (a b)
  (set a (list b))
  )

; ** Indeed an unusual way to obtain a list of digits, but
; ** necessary because of the unusual "things" that result
; ** when an atom is exploded.
(setq digits (cdr (explode 'x0123456789)))


(setq in ())
(setq for ())
(setq inf 'infinity)
(setq minf 'negative_infinity)
```

A.1 Qualitative Simulation of the AEROBEE Controller

```
;                              Aerobee.part3
```

```
;* establishes values of program constants
(defun set_constants ()
  (msetq 'ra 0)
  (msetq 'inertia 900)
  (msetq 'ma 12.5)
  (msetq 'fa 4)
  (msetq 'g1 0.13)
  (msetq 'g2 11.9)
  (msetq 'dead_space 0.025)
  (msetq 'r1 33000)
  (msetq 'r2 33000)
  (msetq 'r3 25000)
  (msetq 'pi 3.14159)

  (setq rad_conversion (qual_divide pi '(180)))
  (setq degree_conversion (qual_divide '(180) pi))

  (setq c1 (qual_divide '(1) (qual_plus '(1) (qual_times (qual_plus
'(1) (qual_divide r3 r2))(qual_divide r1 r3)))))
  (setq c2 (qual_divide '(1) (qual_plus '(1) (qual_times (qual_plus
'(1) (qual_divide r3 r1))(qual_divide r2 r3)))))
  )


; ** Allocates arrays. Sets initial count values, endogenous
; ** variable list, landmarks, maximum number of states,
(defun main ()
  (set_constants)
  (define_symbolic_a)
  (setq endogenous '( (x 2 1) ))
  (setq  landmarks '(0))
  (setq max_states 100)
  (setq acount 1)
  (setq bcount 0)
  (setq old_acount 0)
  (setq old_bcount 0)
  (declare_transitions_array 'atransitions)
  (declare_transitions_array 'btransitions)
  (declare_states_array 'astates)
  (declare_states_array 'bstates)
  (declare_one_state_array 'new_z_states)
  (declare_one_state_array 'new_b_states)
  (declare_one_state_array 'new_non_z_states)
  (declare_one_state_array 'state_candidate)
```

```
  (setq (aref astates 0 0 0) '((0 o) (infinity c)))
  (setq (aref astates 0 0 1) 0)
  (setq (aref astates 0 0 2) '((negative_infinity c) (infinity c)))
  )


; ** A state computations.
  (setq a_computations '( (setq ev (qual_times g2 (qual_sin (qual_times
(qual_minus ra x0)
                                                rad_conversion))))
   (setq fv (qual_minus '(0) (qual_times g1 x1)))
   (setq v (qual_plus (qual_times c1 ev) (qual_times c2 fv)))
   (setq h (dsp v (qual_minus '(0) dead_space) dead_space))
   (setq force (fsw h (qual_minus '(0) fa) '(0) fa))
   (setq x2 (qual_times (qual_divide (qual_times force ma)
                                     inertia) degree_conversion))
))

; ** B state computations.
(setq b_computations '(
   (setq x0 (qual_plus x (qual_times dt x1)))
   (setq x1 (qual_plus x1 (qual_times x2 dt)))
   ))
```

Appendix B

The WP Transform applied to the Aerobee Controller

```
;                                Aerobee.wp.lisp

{The weakest precondition for an observed value is obtained from
the loop of the aerobee rocket simulation. (One constant representing
the mistrusted component is not specified.) Then simplification of
the resulting weakest precondition is obtained using a recursive
descent parser to perform the actual simplifications.
Then an actual numerical value or range of values for the
untrusted variable is searched for using a mathematical technique.
Beginning with a possible interval, endpoints are found by
successively halving the interval and noting the resulting value
this produces in the weakest precondition, terminating when the
weakest precondition evaluates to true.

TO RUN:
  Modify constant values and program specification to represent the
desired program, leaving out a constant value for the mistrusted
component. Specify the desired result predicate in the
"weakest_precondition" function.  Modify the function "try" by
specifying the "unknown", "value" (as the midpoint of), "high" and
"low" endpoints.
  Compile.
  Type (ultimate).
  If only the weakest precondition is desired,
type (weakest_precondition).  If simplified weakest precondition
is desired, run (weakest_precondition) and (simplification).}
```

```
;seg segregates expressions.  It extracts all numeric operands from
;an expression that has the same operator throughout its top level.
;The operation implied by the operator is applied to all the numeric
;operands in the expression.  The result of this is combined with
;any non-numeric operands to be returned as the result of seg.  If
;all top-level operators are not identical or there are no operators
;in the expression, it is returned unchanged.
(defun seg (x &aux temps tempresult)
  (cond
    ((null x) ())
    ((atom x) x)
    ((equal (length x) 1) (seg (car x)))
    ((same_op (cadr x) (cdddr x))
     (setq temps *s*)
     (setq *s* (combine (cadr x) (numbersonleft x)))
     (setq tempresult (app *s*))
     (setq *s* temps)
     tempresult
     )
    (t x)
    )
  )

;app accepts an infix expression as its argument and evaluates it
;left to right (no precedence) until a non-numeric operand is
;encountered.
(defun app (x)
   (cond
     ((null x ) ())
     ((atom x) x)
     ((equal (length x) 1) (car x))
     ((and (numberp (car x)) (numberp (caddr x)))
      (app (cons (apply (cadr x)(list (car x) (caddr x))) (cdddr x)))
      )
     (t x)
     )
   )

;same_op returns t if all operands at the top level of the expression
;y are equal to x.  y is an infix expression with the first element
;removed.
(defun same_op (x y)
```

```
 (cond
   ((null y) t)
   ((equal (car y) x) (same_op x (cddr y)))
   (t ())
   )
 )

;combine returns y with x between each of its elements.  If only one
;element in y, it is returned unchanged.  This is useful for creating
;infix expressions out of a list of operands.
(defun combine (x y)
  (cond
    ((null y) ())
    ((equal (length y) 1) y)
    (t (cons (car y) (cons x (combine x (cdr y)))))
    )
  )
;numbersonleft takes an infix expression as its argument and returns
;a list consisting only of the operands.  Numeric operands precede
;non-numeric ones in the list.
(defun numbersonleft (x)
  (cond
    ((null x ) ())
    ((numberp (car x))
     (cons (car x) (numbersonleft (cddr x))))
    (t
      (append (numbersonleft (cddr x)) (list (car x)))
      )
    )
  )

;symb_times performs symbolic or actual multiplication on its
;arguments.
(defun symb_times (l r)
   (cond
      ((or (safe_zerop l) (safe_zerop r)) 0)
      ((or (eq l 'inf) (eq r 'inf)) 'inf)
      ((eq l 1) r)
      ((eq r 1) l)
      ((and (numberp l) (numberp r)) (* l r))
      ((and (numberp r) (atom l)) (list r '* l))
      ((and (atom l) (atom r)) (list l '* r))
      ((and (listp l)
            (or (equal (length l) 1) (same_op '* (cdr l))))
```

```
          (cond
            ((atom r) (seg (cons r (cons '* l))))
            ((or (equal (length r) 1) (same_op '* (cdr r)))
              (seg (append l (cons '* r)))
            )
            (t (list l '* r))))
        ((and (listp l)
              (numberp (car l))
              (equal (cadr l) '/)
              (numberp r))
         (cons (* r (car l)) (cdr l))
         )
        ((and (listp r)
              (or (equal (length r) 1) (same_op '* (cdr r))))
          (cond
            ((atom l) (seg (cons l (cons '* r))))
            ((or (equal (length l) 1) (same_op '* (cdr l)))
              (seg (append r (cons '* l)))
            )
            (t (list l '* r))))
        ((and (listp r) (equal (cadr r) '/) (numberp (caddr r)))
          (cond
            ((atom l)
             (list
               (seg (cons l (list '* (/ 1 (caddr r)))))
               '* (car r)))
            ((or (equal (length l) 1) (same_op '* (cdr l)))
             (list
               (seg (append l (list '* (/ 1 (caddr r)))))
               '* (car r)))
            (t (list l '* r))
            )
          )
        (t (list l '* r))
    )
)

;safe_zerop is the same as the built-in zero_p, but doesn't crash if
;its argument is not a number.
(defun safe_zerop (x)
  (and (numberp x) (zerop x))
  )

;symb_add performs l+r, either symbolically or numerically, depending
```

```
;on the types of l and r.
(defun symb_add (l r)
  (cond
      ((safe_zerop l) r)
      ((safe_zerop r) l)
      ((or (eq l 'inf) (eq r 'inf)) inf)
      ((and (numberp l) (numberp r)) (+ l r))
      ((and (atom l) (atom r)) (list l '+ r))
      ((and (listp r) (listp l) (numberp (car l)) (numberp (car r))
            (> (car l) (car r)))
       (list r '+ l)
        )
      ((and (listp l)
            (or (equal (length l) 1) (same_op '+ (cdr l))))
      (cond
            ((atom r) (cons r (cons '+ l)))
            ((and (listp r) (numberp (car l)) (numberp (car r))
                  (> (car l) (car r)))
             (list  r '+ l)
             )
            ((or (equal (length r) 1) (same_op '+ (cdr r)))
              (seg (append l (cons '+ r)))
            )
            (t (list l '+ r))))
      ((and (listp r)
            (or (equal (length r) 1) (same_op '+ (cdr r))))
       (cond
            ((atom l) (seg (cons l (cons '+ r))))
            ((or (equal (length l) 1) (same_op '+ (cdr l)))
              (seg (append r (cons '+ l)))
            )
            ((and (listp l) (numberp (car l)) (numberp (car r))
                  (> (car l) (car r)))
             (list r '+ l)
             )
            (t (list l '+ r))))
        (t (list l '+ r))
   )
)

;symb_div returns the actual or symbolic result of l/r.
(defun symb_div (l r)
   (cond
      ((safe_zerop r) 'inf)
```

```
      ((eq l r) 1)
      ((eq l 'inf) 'inf)
      ((eq r 'inf) 0)
      ((eq r 1) l)
      ((and (numberp l) (numberp r)) (/ l r))
      (t (list l '/ r))
   )
)

;symb_minus returns l-r.  This may or may not be an actual numeric
;subtraction, depending on whether l and r are both numeric or not.
(defun symb_minus (l r)
  (cond
      ((eq r l) 0)
      ((and (numberp l) (numberp r)) (- l r))
      ((safe_zerop r) l)
      ((safe_zerop l) (list 'NEG r))
      ((eq l 'inf) 'inf)
      ((eq r 'inf) (simp (list 'NEG 'inf)))
      (t (list l '- r))
   )
)

;symb_neg performs actual or symbolic unary minus on its argument.
(defun symb_neg (x)
  (cond
      ((numberp x) (- 0 x))
      ((atom x) (list 'NEG x))
      ((eq (car x) 'NEG) (cdr x))
      (t (list 'NEG x))
   )
)

;symb_sin returns the sine of its argument if it is numeric.
;Otherwise, (sin x) is returned, where x is the argument.
(defun symb_sin (x)
  (cond
      ((numberp x) (sin x))
      ((atom x) (list 'sin x))
      ((eq (car x) 'sin) (cdr x))
      (t (list 'sin x))
   )
)
```

```
;symb_or performs actual or symbolic manipulations of "l OR r".
(defun symb_or (l r)
  (cond
    ((or (equal l 'T) (equal r 'T)) 'T)
    ((equal l 'F) r)
    ((equal r 'F) l)
    ((and (equal (length l) 3) (equal (length r) 3)
          (equal (car l) (car r)) (equal (cddr l) (cddr r)))
     (cond
       ((and (equal (cadr l) '>=) (equal (cadr r) '>)) l)
       ((or (and (equal (cadr l) '<) (equal (cadr r) '>))
            (and (equal (cadr l) '>) (equal (cadr r) '<)))
        (list (car l) '<> (caddr l)))
       (t (list l 'OR r))
       )
     )

    (t
      (list l 'OR r)
      )
   )
  )

;symb_and performs actual or symbolic manipulations of "l AND r".
(defun symb_and (l r)
  (cond
    ((and (equal l 'T) (equal r 'T)) 'T)
    ((or  (equal l 'F) (equal r 'F)) 'F)
    ((equal l 'T) r)
    ((equal r 'T) l)
    ((and (equal (length l) 3) (equal (length r) 3)
          (equal (car l) (car r)))
     (cond
       ((and (equal (cadr l) '>=)
             (equal (cadr r) '<>)
             (equal (cddr l) (cddr r)))
          (list (car l) '> (caddr l)))
       ((and (or (equal (cadr l) '>) (equal (cadr l) '<))
             (equal (cadr r) '<>) (equal (cddr l) (cddr r)))
        l)
       ((and (or (equal (cadr r) '>) (equal (cadr r) '>))
             (equal (cadr l) '<>) (equal (cddr l) (cddr r)))
        r)
       ((and (equal (cadr l) '>=)
```

```
            (equal (cadr r) '>)
            (numberp (caddr l))
            (numberp (caddr r))
            (null (< (caddr r) (caddr l))))
       r)
      (t (list l 'AND r))
      )
    )
   (t
     (list l 'AND r)
     )
   )
 )

;symb_lt performs actual or symbolic simplification of l < r.
(defun symb_lt (l r)
  (cond
    ((equal (symb_eq l r) 'T) 'F)
    ((and (numberp l) (numberp r))
     (cond
       ((< l r) 'T)
       (t 'F)
       )
     )
    ((and (numberp r) (listp l) (> (length l) 2) (equal (cadr l) '+))
     (cond
       ((numberp (car l))
        (list (cddr l) '< (- r (car l))))
       ((numberp (caddr l))
        (list (car l) '< (- r (caddr l))))
       (t (list l '< r))
       )
     )
     ((and (numberp r) (listp l) (> (length l) 2) (equal (cadr l) '-)
           (numberp (caddr l)))
      (list (car l) '< (+ r (caddr l))))
     (t
       (list l '< r)
       )
     )
   )

 ;symb_eq performs actual or symbolic simplification of l = r.
 (defun symb_eq (l r)
```

```
  (cond
    ((equal l r) 'T)
    ((and (numberp l) (numberp r))
     (cond
       ((< (abs (- l r)) 0.001) 'T)
       (t 'F)
       )
     )
    ((and (listp l) (numberp (car l)) (equal '+ (cadr l))
          (numberp r))
     (list (cddr l) '= (- r (car l))))
    ((and (numberp r) (listp l) (> (length l) 2) (equal (cadr l) '-)
          (numberp (caddr l)))
      (list (car l) '= (+ r (caddr l))))
    (t (list l '= r))
    )
  )

;symb_le performs actual or symbolic simplification of l <= r.
(defun symb_le (l r)
  (list l '<= r)
  )

;symb_ne performs actual or symbolic simplification of l <> r.
(defun symb_ne (l r)
  (cond
    ((equal (symb_eq l r) 'T) 'F)
    ((equal (symb_eq l r) 'F) 'T)
    (t
      (list l '<> r)
      )
    )
  )

;symb_not simplifies "NOT l".
(defun symb_not (l)
  (cond
    ((equal l 'T) 'F)
    ((equal l 'F) 'T)
    ((equal (length l) 3)
     (cond
       ((equal (cadr l) '=) (list (car l) '<> (caddr l)))
       ((equal (cadr l) '<>) (list (car l) '= (caddr l)))
       ((and (equal (cadr l) '<) (numberp (car l)))
```

```
          (list (caddr l) '< (car l)))
         ((equal (cadr l) '<) (list (car l) '>= (caddr l)))
         ((equal (cadr l) '>) (list (car l) '<= (caddr l)))
         ((equal (cadr l) '<=) (list (car l) '> (caddr l)))
         ((equal (cadr l) '>=) (list (car l) '< (caddr l)))
         )
       )
      (t
        (list 'NOT l)
        )
      )
   )

;listp returns t if its argument is a list. () otherwise.
(defun listp (x)
   (null (atom x))
)

;*********************************************************************
;The following functions of the form p_x (where x varies) form a
;recursive descent parser for logical and arithmetic expressions.
;*********************************************************************

(defun p_logterm (&aux logterm)
   (setq logterm (p_logfactor))
   (prog ()
      loop
      (cond
         ((null *s*) (return logterm))
         ((eq (car *s*) 'OR)
              (setq *s* (cdr *s*))
              (setq logterm (symb_or logterm (p_logfactor)))
         )
         (t (return logterm))
      )
      (go loop)
   )
)

(defun p_logfactor (&aux logfactor)
   (setq logfactor (p_boolean))
   (prog ()
      loop
      (cond
```

```
         ((null *s*) (return logfactor))
         ((eq (car *s*) 'AND)
              (setq *s* (cdr *s*))
              (setq logfactor (symb_and logfactor (p_boolean)))
         )
         (t (return logfactor))
      )
      (go loop)
   )
)

(defun p_boolean (&aux boolean)
   (setq boolean (p_expr))
   (prog ()
      loop
      (cond
         ((null *s*) (return boolean))
         ((eq (car *s*) '< )
              (setq *s* (cdr *s*))
              (setq boolean (symb_lt boolean (p_expr)))
         )
         ((eq (car *s*) '>)
              (setq *s* (cdr *s*))
              (setq boolean (symb_le (p_expr) boolean))
         )
         ((eq (car *s*) '<= )
              (setq *s* (cdr *s*))
              (setq boolean (symb_le boolean (p_expr)))
         )
         ((eq (car *s*) '>=)
              (setq *s* (cdr *s*))
              (setq boolean (symb_lt (p_expr) boolean))
         )
         ((eq (car *s*) '= )
              (setq *s* (cdr *s*))
              (setq boolean (symb_eq boolean (p_expr)))
         )
         ((eq (car *s*) '<>)
              (setq *s* (cdr *s*))
              (setq boolean (symb_ne boolean (p_expr)))
         )
         (t (return boolean))
      )
      (go loop)
```

```
   )
)

(defun p_expr (&aux expr)
   (setq expr (p_term))
   (prog ()
      loop
      (cond
         ((null *s*) (return expr))
         ((eq (car *s*) '+)
              (setq *s* (cdr *s*))
              (setq expr (symb_add expr (p_term)))
         )
         ((eq (car *s*) '-)
              (setq *s* (cdr *s*))
              (setq expr (symb_minus expr (p_term)))
         )
         (t (return expr))
      )
      (go loop)
   )
)

(defun p_term (&aux term)
   (setq term (p_factor))
   (prog ()
      loop
      (cond
         ((null *s*) (return term))
         ((eq (car *s*) '*)
              (setq *s* (cdr *s*))
              (setq term (symb_times term (p_factor)))
         )
         ((eq (car *s*) '/)
              (setq *s* (cdr *s*))
              (setq term (symb_div term (p_factor)))
         )
         (t (return term))
      )
      (go loop)
   )
)

(defun p_factor ()
```

```
    (cond
       ((eq (car *s*) 'neg)
            (setq *s* (cdr *s*))
            (symb_neg (p_primary))
       )
       ((eq (car *s*) 'sin)
            (setq *s* (cdr *s*))
            (symb_sin (p_primary))
       )
       ((eq (car *s*) 'NOT)
            (setq *s* (cdr *s*))
            (symb_not (p_primary))
       )
       (t (p_primary))
    )
 )

 (defun p_primary (&aux primary)
    (cond
       ((null *s*) ())
       ((listp (car *s*))
           (setq *s* (append (append (car *s*) (list '*right*)) (cdr
 *s*)))
           (setq primary (p_logterm))
           (setq *s* (cdr *s*))
           primary
       )
       (t
          (setq primary (car *s*))
          (setq *s* (cdr *s*))
          primary
       )
    )
 )
 ;*********************************************************************
 ;some is the same function as described in the experlisp manual --
 ;except this one works!!!
 (defun some (x y)
    (cond
     ((null y) ())
     ((x (car y)) (car y))
     (t (some x (cdr y)))
   )
 )
```

```
(setq printdepth 100)
(setq in ())
(setq for ())

;*********************************************************************
;The next few functions are used for obtaining weakest
;preconditions. (unsimplified)
;*********************************************************************

;sub performs a textual substitution of x for all occurrences of
;z in y (at any level of nesting).
(defun sub (x in y for z)
  (cond
    ((null y) ())
    ((null (atom (car y)))
     (cons (sub x in (car y) for z) (sub x in (cdr y) for z))
     )
    ((eq (car y) z)
     (cons x (sub x in (cdr y) for z))
     )
    (t (cons (car y) (sub x in (cdr y) for z)))
    )
  )

;wp_asn returns wp(s,r) where s is an assignment statement.
(defun wp_asn (s r)
  (cond
    ((> (length (cddr s)) 1)
      (sub (cddr s) in r for (car s))
     )
    (t (sub (caddr s) in r for (car s)))
    )
  )

;wp_seq returns wp(s,r) where s is a sequence of one or more
;statements.
(defun wp_seq (s r)
  (cond
    ((cdr s) (wp (car s) (wp_seq (cdr s) r)))
    (t (wp (car s) r))
    )
  )
```

```
;wp_if returns wp(s,r) where s is an IF statement consisting of one
;or more guarded commands.
(defun wp_if (s r)
  (cond
    ((null s) ())
    (t (if_comb (doif (car s) r) (wp_if (cdr s) r)))
    )
  )

;if_comb is used for combining parts of a wp for an IF.  An AND is
;introduced between each clause.  If there is only one clause, then
;there is no AND.
(defun if_comb (x y)
  (cond
    ((null y) (list x))
    (t (cons x (cons 'AND y)))
    )
  )

;doif performs the actual wp calculation for one branch of an if
;statement.  It also makes use of (p ==> q) == (~p v q).
(defun doif (s r)
  (list (list 'NOT (car s)) 'OR (wp (caddr s) r))
  )

;wp determines what type of statement s is and makes the necessary
;calls to return wp(s,r).
(defun wp (s r)
  (cond
    ((null s) r)
    ((eq (car s) 'IF) (wp_if (cdr s) r))
    ((atom (car s)) (wp_asn s r))
    (t (wp_seq s r))
    )
  )

;*********************************************************************

;plug v in  as the value for the unknown and test it


;do the actual mathematical evaluation
(defun evaluate (l r unknown v oldhighv oldlowv &aux temp1 temp2)
  (setq temp1 (plugandchug l unknown v))
```

```
 (setq *s* r)
 (setq temp2 (plugandchug r unknown v))
 (print (list temp1 temp2 unknown v oldhighv oldlowv))
 (again unknown v oldhighv oldlowv temp1 temp2)
 )

;If the two sides of the equation aren't equal (with a tolerance of
;0.00000001) then halve the interval and try again.
(defun again (unknown v oldhighv oldlowv lres rres)
 (cond
    ((< (abs (- lres rres)) 0.00000001) v)
    ((< lres rres )
        (evaluate (car ans) (cddr ans) unknown
          (/ (+ v oldhighv) 2)  oldhighv v))
    (t
     (evaluate (car ans) (cddr ans) unknown
          (/ (+ v oldlowv) 2)  v  oldlowv )
    )
  )

 (defun plugandchug (l unknown v)
    (setq *s* (sub v in l for unknown))
    (p_logterm)
 )

 (defun findendpoints (low high expr unknown)
   (do ((value low (+ value (abs (/ (- high low) 10)))))
        ( (> value high)  t)
        (add_crossing (plugandchug expr unknown value) value)
       )
   )

 (defun add_crossing (x endpt)
   (cond
     ((or (and (< x 0) (> lastval 0))
          (and (> x 0) (< lastval 0)))
      (print (list 'adding 'crossing 'at lastpt endpt))
      )
     )
    (setq lastval x)
    (setq lastpt endpt)
    )
```

```
(defun set_last_the_first_time (expr unknown val)
  (setq lastpt val)
  (setq lastval (plugandchug expr unknown val))
  )

(defun find_zeros (expr unknown)
  (cond
    ((null crossings) zeros)
    ((< (abs (- (caar crossings)
                (/ (+ (caar crossings) (cadar crossings)) 2)))
        0.001)
     (cons (car crossings) zeros)
     (setq crossings (cdr crossings))
     (find_zeros expr unknown)
     )
    (t
      (set_last_the_first_time expr unknown (caar crossings))
      (findendpoints (caar crossings) (cadar crossings) expr 'r1)
      (setq crossings (cdr crossings))
      (find_zeros expr unknown)
      )
    )
  )



(defun find_values (low high expr unknown)
  (setq zeros ())
  (setq crossings (list (list low high)))
  (find_zeros expr unknown)
  )

;try is a shortcut call to extract a limiting value for r1
(defun try ()
;                              "unknown" "val" "high" "low"
  (evaluate (car ans) (cddr ans)  'r1 500000 1000000 0)
  )
;*********************************************************************
;The following setq's define the simulation program to be used as
;data by this program.
;*********************************************************************

(setq IF2
   '(IF
```

```
      ((h < 0) ==> (f := -4))
      ((h = 0) ==> (f :=  0))
      ((0 < h) ==> (f :=  4))
      )
   )

(setq IF1
   '(IF
      ((v < NEG ds) ==> (h := v + ds))
      ((v <= ds)    ==> (h := 0)     )
      ((ds < v)     ==> (h := v - ds))
      )
   )

;constants
(setq C1 '(ra := 0))
(setq C2 '(i := 900))
(setq C3 '(ma := 12.5))
(setq C4 '(fa := 4))
(setq C5 '(g1 := 0.13))
(setq C6 '(g2 := 11.9))
(setq C7 '(ds := 0.025))
(setq C8 '(r2 := 33000))
(setq C9 '(r3 := 25000))
(setq C10 '(pi := 3.14159))
(setq C11 '(x := 12.0779))
(setq C12 '(x1dot := -1.2732))
(setq C13 '(rc := pi / 180))
(setq C14 '(dc := 180 / pi))

;meta-constants
(setq M1 '(c1 := 1 / (1 + (1 + r3 / r2) * (r1 / r3))))
(setq M2 '(c2 := 1 / (1 + (1 + r3 / r1) * (r2 / r3))))

;statements
(setq S1 '(ev := g2 * sin ((ra - x) * rc)))
(setq S2 '(fv := NEG g1 * x1dot))
(setq S3 '(v  := c1 * ev + c2 * fv))
(setq S4 IF1)
(setq S5 IF2)
(setq S6 '(x2dot := (f * ma / i) * dc))

(setq pgm (list C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14
                M1 M2
```

```
                S1 S2 S3 S4 S5 S6))

;************************************************************************

;obtain the weakest precondition
(defun weakest_precondition ()
  (setq temporary (wp pgm '(x2dot = 3.1831)))
  )

;simplify it
(defun simplification ()
  (setq *s* temporary)
  (p_logterm)
  )

;find the value for r1
(defun ultimate ()
  (weakest_precondition)
  (setq ans (simplification))
  (try)
)
```