# A Fuzzy Controller: Theorems and Proofs about its Dynamic Behavior.

Miren Carranza
William D. Young

Technical Report 91 May 1993

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

## 1. Introduction

The field of *formal methods* attempts to achieve system reliability by applying logic-based mathematical modeling in the construction of digital systems. Using rigorous, mathematically-based techniques that model programs and computing systems as mathematical entities, practitioners of formal methods attempt to prove that program models meet their specifications for all potential inputs. This approach augments more traditional, testing-based software and hardware engineering practice with *deductive* approaches to predicting system behavior and offers promise for enhancing the quality of digital systems, at least in selected applications. A subfield of formal methods is *automated reasoning* which attempts to partially mechanize the process of reasoning about system correctness. Mechanically supported formal methods have been applied to a variety of applications such as: language implementations [13, 17], operating systems [1], concurrent algorithms [8, 11], fault-tolerant systems [3], computer hardware [7], a simple real-time controller [5], and a wide variety of others.

We are attempting to apply such techniques to a new application domain, namely the correctness of fuzzy rule-based control systems. To this end, we have modeled a simple controller in the computational logic of Boyer and Moore [6] and proved some theorems about its behavior using the automatic theorem prover that supports that logic. Our controller is a forward feeding controller which simplifies the simulation of its behavior and lets us concentrate on the verification issues related to fuzzy rule based systems.

We have stated as theorems generalizations about the behavior of the controller. These generalizations (theorems) were made by observations of many simulation runs of the system. Quite a number of our initial generalizations were wrong (weren't theorems). The case for formal proof vs. simulation is discussed. In particular we point out that we can prove statements that generalize the behavior of the system for *n* (finite, unspecified) number of clock ticks.

We also introduce two other controllers that do not satisfy the properties that our original controller was proved to satisfy. The point of this part of the exercise is to illustrate the potential of our verification technology in formally comparing different fuzzy controllers.

In appendix A, we give a brief description of the Boyer-Moore Theorem prover. Appendix B contains the complete formal proof of the lemmas about our controller.

## 2. A Simple Fuzzy Controller

A fuzzy controller properly consists of:

- linguistic variables;
- membership functions (fuzzification functions);
- a rule base;
- a fuzzy rule interpreter; and,
- a de-fuzzification function or functions.

A fuzzy controller operates by sensing a crisp measure, converting it into fuzzy terms and applying the fuzzy rules. The action or actions recommended by the application of the fuzzy rules is then de-fuzzified. An actuator applies the resulting crisp control action.

The rule interpreter as well as the fuzzification and defuzzification functions are general purpose, i.e. they are independent of the particular knowledge base to which they are applied. In what follows we describe our controller's specific knowledge base and the knowledge independent fuzzy interpreter operators.

We have defined a simple fuzzy controller for a water tank with a variable input rate. The point of the controller is to keep the level of water in the tank within a certain intermediate range by adjusting the flow through a drain at the bottom of the tank. Periodically, crisp readings are taken that give the current water level (in the range [0…101]) and water input rate (in the range [0…51]). The water flow is adjusted by opening or closing the drain with settings between 0 and 65. Intuitively, a water input rate of, say, 30 and drain opening of 40 means that water is entering the tank at 30 volume units per time unit and draining at 40 volume units per time unit, allowing the tank level to decrease by 10 volume units per time unit.

To fully define our controller, we must specify the linguistic variables, membership functions, rule base, fuzzy rule interpreter, and de-fuzzification scheme. We explain and illustrate each of these below.

## 2.1 Linguistic Variables and Membership Functions

The linguistic variables for our controller define the various regions for the water level, water input rate, and drain position. These are as follows:

| | |
|---|---|
| water level: | empty, low-level, med-level, high-level, full; |
| input rate: | no-flow, min-flow, med-flow, max-flow; |
| drain position: | closed, min-open, med-open, max-open, wide-open. |

Our membership functions define how particular crisp readings of water level and input rate are mapped into these categories.

We use two special classes of fuzzy numbers to define our membership functions: *triangular fuzzy numbers* and *trapezoidal fuzzy numbers*. A triangular fuzzy number `A` is defined by a triple `(A1, A2, A3)` with membership function:

$M_{tri}(A,x) = 0,$ for $x \le a1,$
$M_{tri}(A,x) = (x - a1)/(a2 - a1),$ for $a1 < x \le a2,$
$M_{tri}(A,x) = (a3 - x)/(a3 - a2),$ for $a2 < x \le a3,$
$M_{tri}(A,x) = 0,$ for $x > a3.$

A trapezoidal fuzzy number is represented by a four-tuple `(A1, A2, A3, A4)`. The membership function of a trapezoidal number is characterized as:

$M_{trap}(A,x) = 0,$ for $x \le a1,$
$M_{trap}(A,x) = (x - a1)/(a2 - a1),$ for $a1 < x \le a2,$
$M_{trap}(A,x) = 1,$ for $a2 < x \le a3,$
$M_{trap}(A,x) = (a4 - x)/(a4 - a3),$ for $a3 < x \le a4,$
$M_{trap}(A,x) = 0,$ for $x > a4.$

The intuition behind these numbers is illustrated in Figure 1. Notice that the membership of `X` in the triangular number is determined by a projection onto the `Y` axis. The closer `X` is to `A2`, the greater its membership in the concept characterized by the fuzzy number. Membership is defined similarly for the trapezoidal number. Notice that `Y` has membership 1 in the concept since it falls into the plateau region between `A2` and `A3`.

For our controller, we define water level via trapezoidal numbers as follows:

| | |
|---|---|
| empty: | (0, 1, 2, 5) |
| low-level: | (3, 10, 30, 40) |
| med-level: | (30, 40, 60, 70) |
| high-level: | (60, 70, 90, 97) |
| full: | (95, 96, 100, 101) |

Thus, a crisp water level reading of 4 falls within the `EMPTY` region to some degree and within `LOW-LEVEL` to

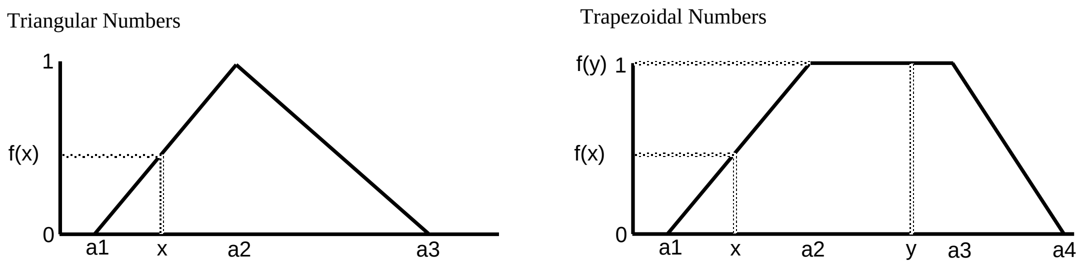

**Figure 1:** Fuzzy Numbers

some other degree. Water input rate is defined via triangular numbers as follows:

| | |
|---|---|
| no-flow: | (0, 1, 3) |
| min-flow: | (2, 15, 25) |
| med-flow: | (20, 30, 35) |
| max-flow: | (30, 40, 51) |

Defining our regions by triangles and trapezoids has a slightly odd effect at the boundaries. For example, a crisp water level value of 0 is *less empty* than a value of 2. We reconcile this by imagining that the actual bottom of the tank is at water level 2 and that no crisp values of water level below 2 are actually encountered.[1]

Now, given a pair of crisp values of water level we determine to what degree it belongs to each of the defining regions using the trapezoidal number and the membership function $\mathtt{M}_{\mathtt{TRAP}}$.[2] For a given crisp water level reading, the degree of membership is determined in each of the various water level regions by applying the $\mathtt{M}_{\mathtt{TRI}}$ membership function defined above. The result is a collection of fuzzy values listing to what degree the crisp reading falls into each of the regions. For example, for a crisp water level of 4 we have the following set of fuzzy numbers.

{(empty 33), (low-level 14), (med-level 0), (high-level 0), (full 0)}

We perform a similar computation on our crisp water input rate, using the trapezoidal numbers defining the various input rate regions. We ignore any values for which the degrees are zero. Thus, for example, a crisp water level of 4 and input rate of 22 yields the following sets:

| | |
|---|---|
| water level: | {(empty 33), (low-level 14)} |
| input rate: | {(min-flow 30), (med-flow 20)} |

This means that a water level of 4 is in region `EMPTY` with degree 33 and in region `LOW-LEVEL` with degree 14. The crisp input rate of 22 is in the `MIN-FLOW` range with degree 30 and in the `MED-FLOW` range with degree 20.

We now combine all of the various possible fuzzy states corresponding to the single crisp input state.

```
{ {(empty 33), (min-flow 30)}
  {(empty 33), (med-flow 20)}                                  (*)
  {(low-level 14), (min-flow 30)}
  {(low-level 14), (med-flow 20)} }
```

[1]We could have used negative values to represent these boundary values. However, for certain technical reasons related to the Boyer-Moore logic, we wished to avoid dealing with negatives in our formalization.

[2]We are using natural number values in the range [0…100] to represent fuzzy numbers, rather than real numbers in the range [0…1]. Thus actually the numbers computed by our membership functions are scaled by 100. This is required due to the difficulty of dealing with reals in the Boyer-Moore logic. The fuzzy value, say, 52 in our representation represents the usual fuzzy value of 0.52.

This list of fuzzy states will determine the selection of rules to apply at the next step.

## 2.2 Controller Rules

In the control world it is desirable that the rules form a *complete* system. This ensures that for any process conditions (within the considered universe) the controller can compute a meaningful control action. To build a complete rule system for a fuzzy controller is feasible.

In a number of cases, completeness is achieved by having the input (sensed) variables assuming all possible values and the rules considering all the combinations. This is achieved in fuzzy controllers by abstracting the possible values in linguistic variables such as ‘‘high,’’ ‘‘low,’’ etc. A fuzzy system with, for example three input variables, each of them with three values (min, med, max) would require $3^3$ rules to describe all possible process conditions.

In our particular controller there are $(5 \times 4)$ process conditions, given that we have two process variables (water level and water input rate) with five and four possible values, respectively. A twenty rule system will cover the system, if there are no duplicate rules.

For our system, a rule is a triple of the form:

<water-level, water-input-rate> $\rightarrow$ control-action

Our controller is specified by the following rules:

| | | | | | |
|---|---|---|---|---|---|
| <full, no-flow> | $\rightarrow$ | wide-open; | <full, min-flow> | $\rightarrow$ | wide-open; |
| <full, med-flow> | $\rightarrow$ | wide-open; | <full, max-flow> | $\rightarrow$ | wide-open; |
| <high-level, no-flow> | $\rightarrow$ | med-open; | <high-level, min-flow> | $\rightarrow$ | med-open; |
| <high-level, med-flow> | $\rightarrow$ | max-open; | <high-level, max-flow> | $\rightarrow$ | wide-open; |
| <med-level, no-flow> | $\rightarrow$ | closed; | <med-level, min-flow> | $\rightarrow$ | min-open; |
| <med-level, med-flow> | $\rightarrow$ | med-open; | <med-level, max-flow> | $\rightarrow$ | max-open; |
| <low-level, no-flow> | $\rightarrow$ | closed; | <low-level, min-flow> | $\rightarrow$ | closed; |
| <low-level, med-flow> | $\rightarrow$ | min-open; | <low-level, max-flow> | $\rightarrow$ | min-open; |
| <empty, no-flow> | $\rightarrow$ | closed; | <empty, min-flow> | $\rightarrow$ | closed; |
| <empty, med-flow> | $\rightarrow$ | closed; | <empty, max-flow> | $\rightarrow$ | closed. |

In the previous section, we illustrated how we derive a collection of the fuzzy process states from crisp inputs. We must now determine to what extent each of our rules fires, given these fuzzy states. To do this, we establish a weighting factor (firing strength) of each rule and then combine the rules according to this weighting factor to determine a final crisp action. Various approaches to this problem are discussed in [10].

To determine the firing strength of each rule we use a method derived independently by Mamdani [12] and Togai [16]. This uses the multivalued logical-implication operator: *truth {a(i) -> b(j)} = min {a(i), b(j)}*. For each rule in our database we determine the activation degree of the consequent—the recommended control action—as follows. The weighting or ‘‘activation value’’ $w(i)$ of the $i^{th}$ rule’s consequent is the *minimum* of its antecedents’ values. For example, suppose we have the following rule:

<low-level, med-flow> $\rightarrow$ min-open.

Suppose also that the determined degree of *low*-ness of the water-level is 14 and the degree of *medium*-ness of water input is 20. In our formulation, this is represented by the fuzzy state

{(low-level 14), (med-flow 20)}

The consequent of this rule, namely `MIN-OPEN`, will be assigned a strength of $min(14, 20) = 14$, and the firing strength of the recommended control action—set the drain at *minimally open*—will have a strength of 14.

There are several methods to combine the different fuzzy control actions considering the corresponding strengths with which each was recommended. We use the de-fuzzification method known as the Fuzzy Centroid Method [9].

Our crisp input values have been fuzzified into a collection of fuzzy states such as `{(LOW-LEVEL 14), (MED-FLOW  20)}`. However, we have designed our rule set in such a way that there is exactly one rule with antecedent `<LOW-LEVEL, MED-FLOW>`, namely,

<low-level, med-flow> $\rightarrow$ min-close.

Thus we can easily select those rules that fire simply by inspecting our collection of fuzzy states.[3] Now, for the list of fuzzy states labeled (*) above, we extract the corresponding rules:

| | | |
|---|---|---|
| <empty, med-flow> | $\rightarrow$ | closed, |
| <empty, min-flow> | $\rightarrow$ | closed, |
| <low-level, med-flow> | $\rightarrow$ | min-open, |
| <low-level, min-flow> | $\rightarrow$ | closed. |

From these rules, we assign to each consequent the minimum weight of its antecedents. This yields the list:

| | | |
|---|---|---|
| closed | with degree 30, | |
| closed | with degree 20, | (**) |
| closed | with degree 14, | |
| min-open | with degree 14. | |

Now to select the final crisp action, we use the *fuzzy centroid* method on this list.

We define the regions for the control action, in this case *drain opening*, by a collection of triangles similar to those defining water input rate.

| | |
|---|---|
| closed: | (0, 1, 3) |
| min-open: | (1, 15, 25) |
| med-open: | (20, 30, 35) |
| max-open: | (30, 40, 50) |
| wide-open: | (45, 55, 65) |

The centroid $C_{region}$ of a triangle is the ‘‘peak’’ of the triangle, e.g., 15 for the `MIN-OPEN` region and 40 for the `MAX-OPEN` region. The final actuator value is determined by the following formula. If we have a set of selected consequents, {$region_i$ with degree $d_i$}, we compute the final value as the floor of:

$$\frac{\Sigma(d_i \times C_{region_i})}{\Sigma(d_i)}$$

To illustrate, for the collection of weighted consequents in (**) above, the final actuator value computed by the fuzzy controller is 3, computed from:

$$\frac{(30\times1)+(20\times1)+(14\times1)+(14\times15)}{(30+20+14+14)}$$

This number is interpreted as the crisp value for the drain setting recommended by the controller. In our formalization, this entire computation is encapsulated into a function called `FUZZY-CONTROLLER`.

## 3. Verifying the Fuzzy Controller

Given the formalization of the fuzzy controller in the preceding section, it remains to specify its correct behavior. Ideally, we would like to assert and prove that the execution of the controller over time maintains the water level of the tank within an acceptable range. To state this formally requires modeling the behavior of the tank under control of the system, i.e., modeling the *environment* in which the controller operates.

[3]Ostensibly, all rules fire. However, most rules have zero weight and will not affect the computation; these can be ignored.

For our controller, the environment is particularly simple. The water input rate is measured at the pipe's opening which is above our tank. The water will reach the tank after a time unit. This kind of controller is known as a *forward feeding controller* and it represents the simplest kind of controller to simulate. Given a current input rate and level, we can accurately predict the effect of the controller on the water level. In particular, we know that the change in level of the tank in a time unit is a very simple function `WATER-FILL` of the current level, water input rate, and water drain rate. We encapsulate this in the function `WATER-FILL`.

water-fill (current-level, water-in, water-out) ≡ (current-level + water-in) - water-out.

We can easily see that if the drain opening value is greater than the water input rate, then the water level in the tank will decrease; if the opening is smaller than the input rate, the level will increase. These are captured in the following three simple theorems.

water-out < water-in $\rightarrow$ level < water-fill (level, water-in, water-out);

water-in = water-out $\rightarrow$ water-fill (level, water-in, water-out) = level;

water-out ≥ water-in $\rightarrow$ level ≥ water-fill (level, water-in, water-out).

We can formalize the simulation of our controller over time via functions in the Boyer-Moore logic. One function *water-fill-overtime* simulates the behavior of the level of water in the tank for an arbitrary sequence of water input rates. However, we are mainly interested in proving properties about the behavior of our system when water input rate is held constant. The function, *behavior-trace* carries out such a simulation and returns a list of successive water level values for *n* time units, where the drain flow is adjusted at each time unit according to the (then) current water level.

```
    behavior-trace (n, wl, rate)
          ≡
    if n = 0,      nil;
    otherwise,     cons (new-wl, behavior-trace (n - 1, new-wl, rate)),
where
    new-wl = water-fill (wl, rate, fuzzy-controller (wl, rate)).
```

The function call *cons (e, l)* adds element *e* to the front of list *l*; *nil* is the empty list.

Asserting that our controller stabilizes the water level in the tank is equivalent to asserting that the behavior trace exhibits certain characteristics. We discuss this further below.

Because functions in the Boyer-Moore logic are executable, we can ''run'' our `BEHAVIOR-TRACE` function for specific values and observe the trace. Some sample runs are illustrated below:

```
behavior-trace (20, 1, 30)  ⇒  [30 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45]

behavior-trace (20, 30, 45)  ⇒  [60 65 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63]

behavior-trace (20, 97, 35)  ⇒  [77 57 52 47 42 37 42 37 42 37 42 37 42 37 42 37 42 37 42 37]
```

## 3.1 From Behaviors to Theorems

In our simulations of the system using the `BEHAVIOR-TRACE` function and others, we observed various patterns in the system behavior and conjectured that the following were true.

- If the tank is initially full, or nearly full, the controller adjusts the drain in such a way that the water level decreases.
- If the tank is initially empty or nearly empty, then the water level tends to increase.

- If the water input rate remains constant for several cycles the water level either stabilizes at a single point within the desired medium range or cycles among several values within the desired range.

We formalized the observed behaviors in theorems in the logic stating precisely the conditions under which these behaviors occur.

1. If the water level is in the empty region, then the level rises *for all* input rates above 2. (Recall that a water input rate of 2 corresponds to no input.)
2. If the water level is within the full region, then the water level decreases *for all* possible values of water input rate.
3. Independently of the original water level, if the water input rate remains constant and within the ranges of 10 and 45, then the water level will either stabilize at a level within the desired boundaries, or the level will cycle among values within the desired boundaries.

We stated each of these conjectures within the Boyer-Moore logic and proved each of them using the Boyer-Moore theorem prover.

To illustrate, the third theorem is expressed in the logic as:[4]

```
(prove-lemma controller-stability-0-100 (rewrite)
   (let ((trace (behavior-trace 30 wl wir)))
      (implies (and (in-intervalp wl (cons 0 100))
                    (in-intervalp wir (cons 10 45)))
               (trace-ok trace 5))))
```

where the function **trace-ok** is defined as

```
trace-ok (trace n)
 (let ((nthcdr (nthcdr n trace)))
   (and (repeats* nthcdr)
        (all-in-desired-levelp nthcdr))))
```

The hypotheses of the theorem assert that the current water level has a value in the appropriate range [0…100] and water input rate is in the medium range of [10…45]. The conclusion of the theorem is a conjunction of two properties encompassed in the definition of *trace-ok*. The first property (defined by **repeats ***) asserts that the trace enters a cycle within at most 5 ‘‘ticks’’ of the clock. The second (defined by **all-in-desired-level** asserts that each of the elements of this cycle are within the desired medium water level range of 30 to 70.

## 3.2 Verification vs. Simulation

Strictly speaking, the theorem *controller-stability-0-100* applies only to behavior traces of length 30. It is possible through exhaustive simulation try all the possible inputs, run each simulation for 30 ticks and observe the results. However, we would like to generalize our result to traces of length *n*.

The following theorem:

```
(prove-lemma all-in-range (rewrite)
  (let ((trace (behavior-trace n wl ir))
        (trace2 (behavior-trace p wl ir)))
    (implies (and (lessp m n)
                  (not (lessp p n))
                  (trace-ok trace m))
             (trace-ok trace2 m))))
```

[4]See Appendix B for the full formalization.

Establishes that if any trace of length *n* has the desired properties defined by *trace-ok* then any longer trace will have those properties too. This says that we can generalize our previous theorem to establish an inductive property.

We need induction to prove this theorem. This theorem represents the kind of assurance that we can not achieve with simulation alone. Of course, with either simulation or formal verification, we are establishing the correctnes of our controller *with respect to a model* of the world in which it operates. Formal verification gives us a tool to prove the intuitions about the system's behavior that were aquired via simulation.

## 4. Different Controllers and Failed Proofs

A different set of rules could have been defined to realize our controller. In what follows we present two other controllers which differ from our original controller in their rules. This is, the membership functions, and the fuzzification, defuzzification and rule intersection methodologies are the same.

### 4.1 Controller A

The following rules characterize a controller whose rules do not differ very much from our original controller (Only the control actions in the rules concerning *full* were changed):

| | | | | | |
|---|---|---|---|---|---|
| <full, no-flow> | → | min-open; | <full, min-flow> | → | med-open; |
| <full, med-flow> | → | max-open; | <full, max-flow> | → | wide-open; |
| <high-level, no-flow> | → | min-open; | <high-level, min-flow> | → | med-open; |
| <high-level, med-flow> | → | max-open; | <high-level, max-flow> | → | wide-open; |
| <med-level, no-flow> | → | closed; | <med-level, min-flow> | → | min-open; |
| <med-level, med-flow> | → | med-open; | <med-level, max-flow> | → | max-open; |
| <low-level, no-flow> | → | closed; | <low-level, min-flow> | → | closed; |
| <low-level, med-flow> | → | min-open; | <low-level, max-flow> | → | min-open; |
| <empty, no-flow> | → | closed; | <empty, min-flow> | → | closed; |
| <empty, med-flow> | → | closed; | <empty, max-flow> | → | closed. |

This controller satisfies all the theorems that our original controller satisfies up to the *Controller-Stability-0-100* theorem. When the theorem prover attempts to prove this theorem it derives the following case which it simplified to *FALSE*. The following is copied from the theorem prover's output:

```
      (IMPLIES (EQUAL WL 5)
               (NOT (EQUAL WIR 35))).

This again simplifies, trivially, to the new conjecture:

      F.

Need we go on?

************** F  A  I  L  E  D **************
```

When we run our simulation of the controller for 30 clock ticks, we observe the following trace for the specific case of a constant input rate of 35 and, an initial water level of 5:

```
*(behavior-trace 30 5 35)
 '(5 10 15 20 25 30 35 35 35 35 35 35 35 35 35 35 35 35 35
     35 35 35 35 35 35 35 35 35 35 35)
```

We can observe that in this case the controller does stabilize in the desired range, but not in 5 ticks or less as our original controller did.

### 4.2 Controller B

Controller B is characterized by the same membership functions as our original controller but the rules are very different (almost the opposite):

| | | | | | |
|---|---|---|---|---|---|
| <full, no-flow> | → | min-open; | <full, min-flow> | → | min-open; |
| <full, med-flow> | → | min-open; | <full, max-flow> | → | min-open; |
| <high-level, no-flow> | → | closed; | <high-level, min-flow> | → | closed; |
| <high-level, med-flow> | → | closed; | <high-level, max-flow> | → | closed; |
| <med-level, no-flow> | → | closed; | <med-level, min-flow> | → | closed; |
| <med-level, med-flow> | → | closed; | <med-level, max-flow> | → | closed; |
| <low-level, no-flow> | → | max-open; | <low-level, min-flow> | → | max-open; |
| <low-level, med-flow> | → | max-open; | <low-level, max-flow> | → | wide-open; |
| <empty, no-flow> | → | min-open; | <empty, min-flow> | → | min-open; |
| <empty, med-flow> | → | med-open; | <empty, max-flow> | → | max-open. |

This is in fact quite a bad controller. This controller does not even satisfy the property that states that the water level rises if the water level is in the empty region and the input water rate is positive.

The developments in fuzzy technology have permitted us to deal with a domain independent interpreter. But having formalized the interpreter operators *in the logic* lets us experiment with different knowledge bases for the particular domain (i.e., a fluid system in this case). We have shown that we can use the same proof system to analyze the properties of controllers that have the same control goal but differ in their knowledge bases. In the next section we elaborate on future work based on this approach.

## 5. Conclusions and Future Directions

We have modeled a fuzzy controller within a formal logic and proved several properties of the system. The controller is quite simple. However, we believe that the exercise was a useful experiment in applying formal methods to a new and important area. The behavior of controllers in general, and fuzzy controllers in particular, is subtle. Stating formally and proving rigorously behavioral properties of such controllers is one way to gain significantly enhanced assurance of their correctness.

We also discussed two other controllers, built for the same purpose, but which did not satisfy the properties of the original controller. This illustrates that we can use our approach to formally compare different controllers. Though we did not have theorems explicitly relating two different controllers, in attempting to prove the same theorems for a different knowledge base the theorem prover found counterexamples. We believe that our work, though preliminary, points the way to an approach that can be applied to more complex controllers.

We would like to extend this work in several ways.

- The mechanical support in the prover for the underlying mathematics of traditional mathematical modeling and of fuzzy control are lacking. Rationals are supported in the Boyer-Moore logic, but real number arithmetic and analysis is not. Advanced work in this area may require using an alternative proof system. However, we hope to continue with the present tools and develop as much proof support as possible within our current proof paradigm.

- Fuzzy controllers differing in their:
    - membership-functions,
    - rule-intersection-methods, and
    - defuzzification techniques

  can be designed to perform the same control task. The question arises: which of the controllers behaves better?, where "better" is defined as some property. We could prove statements that formally compare the behavior of same-goal/different-implementation controllers with respect to the same model. This could be very useful in determining the best fuzzy control techniques to implement a controller. A simulation-based approach for determining dynamically (at each point in time) which controller is better (the controllers differing on their interpreter functions) can be found in [14]. We barely touched this point of comparing controllers with our discussion of controllers A and B, we would like however, to prove statements explicitly referencing the different controllers. In the case of the controllers A and B the knowledge base was changed but not the interpreter operators.

- Our controller was a very simple controller. A benchmark problem in control theory seems to be the inverted pendulum problem. We would like to formalize a fuzzy controller and model for the inverted pendulum. This is a very complex controller. To model the behavior of this controller requires analytical extensions to the theorem prover. We would like though, to pursue the necessary research to be able to prove statements about the behavior of an inverted pendulum.

**Acknowledgements:** We want to acknowledge the work of the many researchers at Computational Logic who have contributed to the development of proof methodologies and strategies that have paved the way for the formalization and verification of systems in the Boyer-Moore logic. In particular, we appreciate the technical help of J S. Moore and Matt Kaufmann. Miren Carranza wants to express her appreciation to Warren A. Hunt, Jr., Michael K. Smith and Robert S. Boyer for their encouragement she received from them on this project. Also she thanks Dr. Benito Fernandez of the University of Texas at Austin for his help on the topic of fuzzy systems.

# Appendix A
# The Boyer-Moore Logic and Theorem Prover

The Boyer-Moore logic [4, 6] is a quantifier-free, first-order predicate calculus with equality and induction. Logic formulas are written in a prefix-style, Lisp-like notation. Recursive functions may be defined and must be proven to terminate. The logic includes several built-in data types: Booleans, natural numbers, lists, literal atoms, and integers. Additional data types can be defined. The syntax, axioms, and rules of inference of the logic are given precisely in *A Computational Logic Handbook* [6].

The Boyer-Moore logic can be extended by the application of the following axiomatic acts: defining functions, adding recursively constructed data types, and adding arbitrary axioms. Adding an arbitrary formula as an axiom does not guarantee the soundness of the logic; we do not use this feature.

The Boyer-Moore theorem proving system (theorem prover) is a Common Lisp [15] program that provides a user with various commands to extend the logic and to prove theorems. A user enters theorem prover commands through the top-level Common Lisp interpreter. The theorem prover manages the axiom database, function and data type definitions, and proved theorems, thus allowing a user to concentrate on the less mundane aspects of proof development. The theorem prover contains a simplifier and rewriter and decision procedures for propositional logic and linear arithmetic. It also can perform structural inductions automatically.

The Boyer-Moore system also contains an interpreter for the logic that allows the evaluation of terms in the logic. Thus, the logic can be considered as either a functional programming language or an executable specification language. We often use this facility for debugging our specifications.

The Boyer-Moore prover has been used to check the proofs of some quite deep theorems. For example, some theorems from traditional mathematics that have been mechanically checked using the system include proofs of: the existence and uniqueness of prime factorizations; Gauss' law of quadratic reciprocity; the Church-Rosser theorem for lambda calculus; the infinite Ramsey theorem for the exponent 2 case; and Goedel's incompleteness theorem. Somewhat outside the range of traditional mathematics, the theorem prover has been used to check: the recursive unsolvability of the halting problem for Pure Lisp; the proof of invertibility of a widely used public key encryption algorithm; the correctness of metatheoretic simplifiers for the logic; the correctness of a simple real-time control algorithm; the optimality of a transformation for introducing concurrency into sorting networks; and a verified proof system for the Unity logic of concurrent processes. When connected to a specialized front-end for Fortran, the system has also proved the correctness of Fortran implementations of a fast string searching algorithm and a linear time majority vote algorithm. Recent work at CLI includes the mechanically checked proofs of a high-level language compiler, an assembler, a microprocessor, and a simple multi-tasking operating system. These verified components were integrated into a *vertically verified system* called the ‘‘CLI Short Stack’’ [2], the first such system of which we are aware. Many other interesting theorems have been proven as well. It is important to note that all of these proofs were checked by the same general purpose theorem prover, not a number of specialized routines optimized for specific problems.

# Appendix B
## The Fuzzy Controller Event List

```
(boot-strap nqthm)

;; Some basic arithmetic facts:
(prove-lemma plus-right-id2 (rewrite)
  (implies (not (numberp y))
           (equal (plus x y)
                  (fix x))))

(prove-lemma commutativity2-of-plus (rewrite)
  (equal (plus x (plus y z))
         (plus y (plus x z))))

(prove-lemma commutativity-of-plus (rewrite)
  (equal (plus x y)
         (plus y x)))

(prove-lemma plus-equal-0 (rewrite)
  (equal (equal (plus a b)
                0)
         (and (zerop a)
              (zerop b))))

(prove-lemma difference-plus (rewrite)
  (and (equal (difference (plus x y) x)
              (fix y))
       (equal (difference (plus y x) x)
              (fix y))))

(prove-lemma difference-0 (rewrite)
  (implies (not (lessp y x))
           (equal (difference x y) 0)))

(prove-lemma times-zero2 (rewrite)
  (implies (not (numberp y))
           (equal (times x y)
                  0)))

(prove-lemma times-add1 (rewrite)
  (equal (times x (add1 y))
         (if (numberp y)
             (plus x (times x y))
           (fix x))))

(prove-lemma commutativity-of-times (rewrite)
  (equal (times x y)
         (times y x)))

(prove-lemma equal-times-0 (rewrite)
  (equal (equal (times x y)
                0)
         (or (zerop x)
             (zerop y))))

(prove-lemma equal-lessp (rewrite)
  (equal (equal (lessp x y)
                z)
         (if (lessp x y)
             (equal t z)
           (equal f z))))

(prove-lemma difference-elim (elim)
  (implies (and (numberp y)
                (not (lessp y x)))
           (equal (plus x (difference y x))
                  y)))
```

```
(prove-lemma lessp-times-1 (rewrite)
  (implies (not (zerop i))
           (not (lessp (times i j) j))))

(prove-lemma lessp-times-2 (rewrite)
  (implies (not (zerop i))
           (not (lessp (times j i) j))))

(prove-lemma difference-plus1 (rewrite)
  (equal (difference (plus x y) x)
         (fix y)))

(prove-lemma difference-plus2 (rewrite)
  (equal (difference (plus y x) x)
         (fix y)))

(prove-lemma lessp-times-cancellation (rewrite)
  (equal (lessp (times x z)
                (times y z))
         (and (not (zerop z))
              (lessp x y))))

(prove-lemma remainder-quotient (rewrite)
  (equal (plus (remainder x y)
               (times y (quotient x y)))
         (fix x)))

(prove-lemma remainder-wrt-12 (rewrite)
  (implies (not (numberp x))
           (equal (remainder y x)
                  (fix y))))

(prove-lemma lessp-remainder2 (rewrite generalize)
  (equal (lessp (remainder x y)
                y)
         (not (zerop y))))

(prove-lemma remainder-quotient-elim (elim)
  (implies (and (not (zerop y))
                (numberp x))
           (equal (plus (remainder x y)
                        (times y (quotient x y)))
                  x)))

(prove-lemma lessp-times (rewrite)
      (implies (not (zerop x))
               (equal (lessp (times x a) (times x b))
                      (lessp a b))))

(defn lessp-quotient-hint (x y d)
  (if (or (zerop d)
          (lessp x d)
          (lessp y d))
      t
    (lessp-quotient-hint (difference x d)
                         (difference y d)
                         d)))

(prove-lemma lessp-quotient (rewrite)
      (implies (not (lessp x y))
               (not (lessp (quotient x d) (quotient y d))))
      ((induct (lessp-quotient-hint x y d))))

(prove-lemma quotient-times (rewrite)
      (equal (quotient (times a d) d)
             (if (zerop d) 0 (fix a))))
```

```
(prove-lemma f-quotient-aux (rewrite)
      (implies (not (lessp x b))
               (not (lessp 100 (quotient (times 100 b) x))))
      ((disable lessp-times lessp-quotient)
       (use (lessp-times (x 100) (a x) (z b))
            (lessp-quotient (x (times 100 x))
                            (y (times 100 b))
                            (d x)))))

(prove-lemma lessp-plus-difference (rewrite)
   (implies (lessp i x)
            (equal (plus i (difference x i))
                   (fix x))))

;; Some elementary functions.
(defn plistp (x)
  (if (nlistp x)
      (equal x nil)
    (plistp (cdr x))))
(prove-lemma car-append (rewrite)
  (equal (car (append x y))
         (if (listp x)
             (car x)
           (car y))))

(prove-lemma associativity-of-append (rewrite)
  (equal (append (append x y) z)
         (append x (append y z))))

(defn length (x)
  (if (nlistp x)
      0
    (add1 (length (cdr x)))))

(prove-lemma length-positive (rewrite)
  (implies (listp x)
           (lessp 0 (length x))))

(prove-lemma member-append (rewrite)
  (equal (member x (append y z))
         (or (member x y)
             (member x z))))

(defn nthcdr (n list)
  (if (zerop n)
      list
    (nthcdr (sub1 n) (cdr list))))

(defn subset (x y)
  (if (nlistp x)
      t
    (and (member (car x) y)
         (subset (cdr x) y))))


;; Fuzzy logic definitions
(defn memb-fn (l) (car l))

(defn linguistic-variable-name (mfn) (car mfn))
(defn tr (mfn) (cdr mfn))

;; A trapezoidal number is a four-tuple.  These select the components of the
;; number.
(defn a1 (tr) (car tr))
(defn a2 (tr) (cadr tr))
(defn a3 (tr) (caddr tr))
(defn a4 (tr) (cadddr tr))

;; An interval is merely a pair.
(defn interval (a b) (cons a b))

; The system has a tank, a sensor that measures level, a faucet and a
; sensor that measures the rate at which water comes in.
```

```
(defn water-level ()
  ; Defines (coarsely) the regions in bucket
  ’((Empty 0 1 2 5)
    (Low-level 3 10 30 40)
    (Med-level 30 40 60 70)
    (High-level 60 70 90 97)
    (Full 95 96 100 101)))
(defn water-input-rate ()
  ;; Defines the various possible water fill rates
  ’((No-flow 0 1 3)
    (Min-flow 2 15 25)
    (Med-flow 20 30 35)
    (Max-flow 30 40 51)))

;; This defines the drain openings controlling the outflow of water from the
;; tank.  Placing the controller at position 1 means outflow is zero.
;; water out is (drain-opening number) - 1.  If opening is 2 means
;; that water out is 1 unit/time unit
(defn drain-opening () ; Define drain openings
  ’((Closed 0 1 3)
    (Min-open 1 15 25)
    (Med-open 20 30 35)
    (Max-open 30 40 50)
    (Wide-open 45 55 65)))
(defn all-vars ()
   (append (water-level) (append (water-input-rate) (drain-opening))))

(defn names (varlist)
  (if (nlistp varlist)
      nil
    (cons (caar varlist) (names (cdr varlist)))))

(defn water-level-names ()
  (names (water-level)))


;; ’(EMPTY LOW-LEVEL MED-LEVEL HIGH-LEVEL FULL)
(defn water-input-rate-names ()
  (names (water-input-rate)))

;;  ’(NO-FLOW MIN-FLOW MED-FLOW MAX-FLOW)
(defn drain-opening-names ()
  (names (drain-opening)))

;; ’(CLOSED MIN-OPEN MED-OPEN MAX-OPEN WIDE-OPEN)
(defn varp (a)
  (or (member a (water-level-names))
      (member a (water-input-rate-names))
      (member a (drain-opening-names))))
(defn trianglep1 (n)
  ;; a1 < a2 < a3
  (and (lessp  (a1  n) (a2 n))
       (lessp  (a2  n) (a3 n))))

;; A trianglep is of the form (name a1 a2 a3), where
;; name is a varp and (a1 a2 a3) is a triangle number.
(defn trianglep (n)
  (and (equal (length n) 4)
       (varp (linguistic-variable-name n))
       (trianglep1 (tr n))))

(defn triangle-listp (l)
  (if (nlistp l)
      t
    (and (trianglep (car l))
         (triangle-listp (cdr l)))))

;; A trapezoidal number is a four-tuple (a1 a2 a3 a4) with
;; a1 < a2 < a3 < a4
(defn trapezoidp1 (fn)
  (and (lessp  (a1 fn) (a2 fn))
       (lessp  (a2 fn) (a3 fn))
       (lessp  (a3 fn) (a4 fn))))
```

```
(defn trapezoidp (fn)
  (and (equal (length  fn) 5)
       (varp (linguistic-variable-name fn))
       (trapezoidp1 (tr fn))))

(defn trapezoid-listp (fn)
   (If (nlistp fn)
       t
      (and (trapezoidp (car fn))
           (trapezoid-listp (cdr fn)))))

(defn water-level-varp (var)
  ;; This defines the
  (and (trapezoidp var)
       (member (linguistic-variable-name var) (water-level-names))))

(defn water-level-vars-listp (l)
  (if (nlistp l)
      t
    (and (water-level-varp (car l))
         (water-level-vars-listp (cdr l)))))

(defn water-input-rate-varp (var)
  (and (trianglep var)
       (member (linguistic-variable-name var) (water-input-rate-names))))

(defn water-input-rate-vars-listp (l)
  (if (nlistp l)
      t
    (and (water-input-rate-varp (car l))
         (water-input-rate-vars-listp (cdr l)))))

(defn drain-opening-varp (var)
  (and (trianglep var)
       (member (linguistic-variable-name var) (drain-opening-names))))

(defn drain-opening-vars-listp (l)
  (if (nlistp l)
      t
    (and (drain-opening-varp (car l))
         (drain-opening-vars-listp (cdr l)))))

;; Defining the RULE Base
(defn rule-level (rule) (car rule))
(defn rule-flow (rule) (cadr rule))
(defn rule-drain-position (rule) (caddr rule))
(defn rulep (rule)
  (and (equal (length rule) 3)
       (member (rule-level rule)
               (water-level-names))
       (member (rule-flow rule)
               (water-input-rate-names))
       (member (rule-drain-position rule)
               (drain-opening-names))))

(defn rule-listp (rules)
  (if (nlistp rules)
      t
    (and (rulep (car rules))
         (rule-listp (cdr rules)))))

;; It’s the drain and not the flow rate that is being controlled.
```

```
(defn rules ()
  ;; This is the defining set of control rules for our
  ;; simple controller.
  '((full       no-flow   wide-open)
    (full       min-flow  wide-open)
    (full       med-flow  wide-open)
    (full       max-flow  wide-open)
    (high-level no-flow   med-open)
    (high-level min-flow  med-open)
    (high-level med-flow  max-open)
    (high-level max-flow  wide-open)
    (med-level  no-flow   closed)
    (med-level  min-flow  min-open)
    (med-level  med-flow  med-open)
    (med-level  max-flow  max-open)
    (low-level  no-flow   closed)
    (low-level  min-flow  closed)
    (low-level  med-flow  min-open)
    (low-level  max-flow  min-open)
    (empty      no-flow   closed)
    (empty      min-flow  closed)
    (empty      med-flow  closed)
    (empty      max-flow  closed)))

;; Some simple sanity checks for the rules
(prove-lemma rules-ok nil
      (rule-listp (rules)))

(prove-lemma water-level-ok nil
      (water-level-vars-listp (water-level)))

(prove-lemma water-input-rate-ok nil
      (water-input-rate-vars-listp (water-input-rate)))

(prove-lemma drain-opening-ok nil
      (drain-opening-vars-listp (drain-opening)))

;*******  interval arithmetic ****
(defn intervalp (x)                             ; Recognizes intervals
   (and (listp x)
        (numberp (car x))
        (numberp (cdr x))))

(defn interval-add (x y)
   (cons (plus (car x) (car y))
         (plus (cdr x) (cdr y))))

(defn in-intervalp (a x)
  ;; a is in interval x;
  ;; intervals are  open from below.
  (and (lessp (car x) a)
       (not (lessp (cdr x) a))))

(defn interval-diff ( x y)
   (cons (difference (car x) (cdr y))
         (difference (cdr x) (car y))))

(prove-lemma interval-add-comm-aux1 (rewrite)
      (implies (and (in-intervalp a x)
                    (in-intervalp b y))
               (in-intervalp (plus a b) (interval-add x y))))

(disable interval-add-comm-aux1)

(prove-lemma interval-add-commutative (rewrite)
      (implies (and (intervalp x)
                    (intervalp y))
               (equal (interval-add x y)
                      (interval-add y x))))
```

```
(prove-lemma interval-add-associative nil
      (implies (and (intervalp x)
                    (intervalp y)
                    (intervalp z))
               (equal (interval-add (interval-add x y) z)
                      (interval-add x (interval-add y z)))))

(disable interval-add-commutative)


;; ******* FUZZY MEMBERSHIP
;; ** triangle  selectors: a1 a2 a3
;; ** trapezoid selectors: a1 a2 a3 a4
;; ** linguistic variable selector: linguistic-variable-name
;; ** membership function selector: memb-fn
;; ** interval constructor: interval
(defn fuzzyp (n)
  ;; A fuzzy number is simply a number less than or equal to 100
  (not (lessp 100 n)))

(defn fuzz-triangle1 (x a1 a2 a3)
  ;; Given a fuzzy triangle and a value x, this determines to what
  ;; fuzzy value x belongs to the triangle.  It always returns a fuzzy
  ;; value.
  (if (not (lessp a1 x))
      0
    (if (in-intervalp x (interval a1 a2 ))
        (quotient (times 100 (difference x a1) )
                  (difference a2 a1))
      (if (in-intervalp x (interval a2 a3))
          (quotient (times 100 (difference a3 x) )
                    (difference a3 a2))
        0))))
(prove-lemma fuzz-triangle1-fuzzifies nil
  ;; This is an interesting property of our fuzzification
  ;; process, but is not used subsequently.
  (implies (trianglep1 (list a1 a2 a3))
           (fuzzyp (fuzz-triangle1 x  a1 a2 a3))))

(defn fuzz-trapezoid1 (x a1 a2 a3 a4)
  ;; Given a fuzzy trapezoid and a value x, this determines to what
  ;; fuzzy value x belongs to the trap.  It always returns a fuzzy
  ;; value.
  (if (not (lessp a1 x))
      0
    (if (in-intervalp x (interval a1 a2))
        (quotient (times 100 (difference x a1) )
                  (difference a2 a1))
      (if (in-intervalp x (interval a2 a3))
          100
        (if (in-intervalp x (interval a3 a4))
            (quotient (times 100 (difference a4 x))
                      (difference a4 a3))
          0)))))

(prove-lemma fuzz-trapezoid1-fuzzifies nil
  ;; Another interesting property of our fuzzification
  ;; algorithm.
  (implies (trapezoidp1 (list a1 a2 a3 a4))
           (fuzzyp (fuzz-trapezoid1 x a1 a2 a3 a4))))

(defn fuzz-triangle (x mfn)
  ;; Given a value x and list mfn of named fuzzy triangles, this computes
  ;; the value list of named values to which x belongs to each.
  (if (nlistp mfn)
      nil
    (let ((triangle (car mfn)))
      (cons (list (linguistic-variable-name triangle)
                  (fuzz-triangle1 x
                                  (a1 (tr triangle))
                                  (a2 (tr triangle))
                                  (a3 (tr triangle))))
            (fuzz-triangle x (cdr mfn))))))
```

```
(defn fuzz-trapezoid (x mfn)
  ;; Given a value x and list mfn of named fuzzy triangles, this computes
  ;; the value list of named values to which x belongs to each.
  (if (not(listp mfn))
      nil
    (let ((trapezoid (car mfn)))
      (cons (list (linguistic-variable-name trapezoid)
                  (fuzz-trapezoid1 x
                                   (a1 (tr trapezoid))
                                   (a2 (tr trapezoid))
                                   (a3 (tr trapezoid))
                                   (a4 (tr trapezoid))))
            (fuzz-trapezoid x (cdr mfn))))))

(defn fzero-elim (l)
  (if (nlistp l)
      nil
    (if (equal (cadar l) 0)
        (fzero-elim (cdr l))
      (cons (car l)
            (fzero-elim (cdr l))))))

(defn fcombiner1 (a l)
  (if (nlistp l)
      nil
    (cons (list a (car l))
          (fcombiner1 a (cdr l)))))

(defn fcombiner (l1 l2)
  (if (nlistp l1)
      nil
    (append (fcombiner1 (car l1) l2)
            (fcombiner (cdr l1) l2))))

(defn crisp-water-level (process)
   (car process))

(defn crisp-input-rate (process)
   (cdr process))

(defn fuzzifier (process-state)
  (fcombiner (fzero-elim (fuzz-trapezoid (crisp-water-level process-state)
                                         (water-level)))
             (fzero-elim (fuzz-triangle (crisp-input-rate process-state)
                                        (water-input-rate )))))


;; These are selector functions on fuzzy process states which look like:
;;  ((low-level 30) (Med-flow 10))
(defn fprocess-water-level (process-state)
  (car process-state))
(defn fprocess-input-rate (process-state)
  (cadr process-state))
(defn variable-name (pair)
  (car pair))
(defn fuzzy-value (pair)
  (cadr pair))
(defn f-min (fps)
  ;; Notice that this is just MIN specialized to processes
  (if (lessp (fuzzy-value (fprocess-water-level fps))
             (fuzzy-value (fprocess-input-rate fps)))
      (fuzzy-value (fprocess-water-level fps))
      (fuzzy-value (fprocess-input-rate fps))))

; There is only one rule that has a given antecedent i.e., no two different
; control actions correspond to the same antecedent.
; For each fuzzy process state, which matching rule is found.
```

```
(defn antecedent (fps rules)
  ;; Choose the rule that applies.
  (if (nlistp rules)
      nil
    (if (and (equal (rule-level (car rules))
                    (variable-name (fprocess-water-level fps)))
             (equal (rule-flow (car rules))
                    (variable-name (fprocess-input-rate fps))))
        (cons (f-min fps) (rule-drain-position (car rules)))
      (antecedent fps (cdr rules)))))

(defn active-rules (fps rules)
  ;; Choose from the list of rules the ones that apply.
  (if (nlistp fps)
      nil
    (if (antecedent (car fps) rules)
        (cons (antecedent (car fps) rules)
              (active-rules (cdr fps) rules))
      (active-rules (cdr fps) rules))))

(defn f-centroid (action vars)
  (if (nlistp vars)
      nil
    (if (equal action (caar vars))
        (if (equal (length (car vars)) 5)                     ;trapezoid
            (quotient (plus (caddar vars) (cadddar vars))
                      2)
          (caddar vars))                                      ;triangle
        (f-centroid action (cdr vars)))))

(defn weigh-products (vars actions)
  (if (nlistp actions)
      0
    (plus (times (caar actions)
                 (f-centroid (cdar actions) vars))
          (weigh-products vars (cdr actions)))))

(defn sum-weights (actions)
  (if (nlistp actions)
      0
    (plus (caar actions) (sum-weights (cdr actions)))))

(defn fuzzy-controller (process rules) ;vars is not a parameter
  (quotient (weigh-products (all-vars)
                            (active-rules (fuzzifier process) rules))
            (sum-weights (active-rules (fuzzifier process) rules))))

;************ physics *******************


(defn water-fill (level w-i w-o)
  (difference (plus level w-i) w-o))

(prove-lemma level-rises nil
      (implies (lessp w-o w-i)
               (lessp level (water-fill level w-i w-o))))

(prove-lemma water-stays nil
      (implies (equal w-i w-o)
               (equal (water-fill level w-i w-o) (fix level))))
(prove-lemma level-decreases nil
      (implies (not (lessp w-o w-i))
               (not (lessp level (water-fill level w-i w-o)))))

(prove-lemma level-rises1 nil
      (implies (lessp w-o w-i)
               (equal (lessp level (water-fill level w-i w-o))
                      t)))

(prove-lemma level-decreases1 nil
      (implies (not (lessp w-o w-i))
               (equal (lessp level (water-fill level w-i w-o))
                      f)))

;***************simulator**********************
```

```
(defn levelp (x)
  (in-intervalp x (cons 0 101)))

(defn waterinp (x)
  (in-intervalp x (cons 0 51)))

(defn drainopnp (x)
  (in-intervalp x (cons 0 65)))

(defn processp (process)
  (and (listp process)
       (levelp (crisp-water-level process))
       (waterinp (crisp-input-rate process))))

(defn empty-levelp (x)
  (in-intervalp x (cons 0 7)))

(disable water-fill)

(defn water-level-order ()
   '(full high-level med-level low-level empty))

(defn water-input-rate-order ()
   '(max-flow med-flow min-flow no-flow))

(defn number-list (n m)
  (if (lessp n m)
      (cons (add1 n) (number-list (add1 n) m))
    nil)
  ((lessp (difference m n))))

(prove-lemma in-intervalp-rewrite (rewrite)
  (equal (in-intervalp x (cons n m))
         (member x (number-list n m))))

;; This proves but it's rather slow.
(prove-lemma empty-level-rises* ()
  (implies (and (in-intervalp ir (interval 1 51))
                (in-intervalp wl (interval 0 7)))
           (lessp wl (water-fill wl ir
                                 (fuzzy-controller (cons wl ir) (rules)))))
  ((disable in-intervalp)))

(prove-lemma empty-level-rises ()
  ;; Key property showing the behavior of the controller.
  (implies (and (processp process)
                (lessp 2 (crisp-input-rate process))
                (empty-levelp (crisp-water-level process)))
           (lessp (crisp-water-level process)
                  (water-fill (crisp-water-level process)
                              (crisp-input-rate process)
                              (fuzzy-controller process (rules)))))
  ((use (empty-level-rises* (wl (crisp-water-level process))
                            (ir (crisp-input-rate process))))
   (disable fuzzy-controller water-fill in-intervalp-rewrite)))

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

;; At this point we begin the specification and proof of some
;; theorems about the simulation of the system.  In particular,
;; show that if the system ever gets into the intermediate range
;; it will stay in that range.
(defn new-wl (water-level input-rate)
  (water-fill water-level
              input-rate
              (fuzzy-controller (cons water-level input-rate) (rules))))

(defn behavior-trace (n water-level input-rate)
  (let ((new-water-level (new-wl water-level input-rate)))
    (if (zerop n)
        nil
      (cons water-level
            (behavior-trace (sub1 n) new-water-level input-rate)))))
```

```
(defn traces-match (trace1 trace2)
   ;; This says that the traces match element for element
   ;; until one or both of them run out.
   (if (or (nlistp trace1)
           (nlistp trace2))
       t
     (and (equal (car trace1) (car trace2))
          (traces-match (cdr trace1) (cdr trace2)))))

(prove-lemma traces-match-symmetric (rewrite)
  (equal (traces-match x y)
         (traces-match y x)))

(disable traces-match-symmetric)
(prove-lemma behavior-traces-match (rewrite)
      (traces-match (behavior-trace n wl wir)
                    (behavior-trace m wl wir))
      ((disable water-fill fuzzy-controller)))

(prove-lemma traces-match-cars-match (rewrite)
      (implies (and (listp trace1)
                    (listp trace2)
                    (traces-match trace1 trace2))
               (equal (car trace1) (car trace2))))

(disable traces-match-cars-match)
(disable new-wl)
(prove-lemma length-trace (rewrite)
  (equal (length (behavior-trace n wl ir))
         (fix n)))

(defn cyclep (cycle trace)
  ;; trace is cycle cycle cycle... partial_cycle
  (if (listp cycle)
      (if (listp trace)
          (and (equal (car cycle) (car trace))
               (cyclep (append (cdr cycle) (list (car cycle)))
                       (cdr trace)))
        t)
    f))

(prove-lemma trace-hack-1 (rewrite)
  (implies (and (equal (behavior-trace n (new-wl x ir) ir) lst)
                (equal y (car lst)))
           (equal (equal (behavior-trace n y ir) lst) t)))

(prove-lemma trace-hack-2 (rewrite)
  (implies (equal (behavior-trace z (new-wl v ir) ir)
                  (cons w (append d (cons v x))))
           (equal (equal (behavior-trace z w ir)
                         (behavior-trace z (new-wl v ir) ir))
                  t)))

(prove-lemma trace-hack-3 (rewrite)
  (implies (and (not (equal n 0))
                (numberp n)
                (listp stuff)
                (not (listp (cdr stuff)))
                (equal (behavior-trace n (car stuff) ir)
                       (cons (car stuff)
                             (cons (car stuff) (cons v w)))))
           (equal (equal (car stuff) v) t)))

(defn main-trace-property-induction (input ir n initial)
  (if (zerop n)
      nil
    (main-trace-property-induction
     (new-wl input ir) ir (sub1 n) (cdr initial))))

(prove-lemma main-trace-property (rewrite)
  (implies (equal (behavior-trace n a ir)
                  (append initial (cons x (cons y rest))))
           (equal (new-wl x ir) y))
  ((induct (main-trace-property-induction a ir n initial))))
```

```
(prove-lemma trace-hack-4 (rewrite)
  (implies (and (listp stuff)
                (equal (behavior-trace n z ir)
                       (cons z
                             (append stuff
                                     (cons z (cons v w))))))
           (equal (car stuff) v)))

(defn finding-a-cycle-induction (stuff rest ir n)
  (if (zerop n)
      t
    (if (nlistp (cdr rest))
        t
      (finding-a-cycle-induction (append (cdr stuff) (list (car stuff)))
                                 (cdr rest)
                                 ir (sub1 n)))))

(prove-lemma finding-a-cycle (rewrite)
  (implies (and (equal (behavior-trace n (car stuff) ir)
                       (append stuff rest))
                (listp stuff)
                (listp rest)
                (equal (car stuff) (car rest)))
           (cyclep stuff rest))
  ((induct (finding-a-cycle-induction stuff rest ir n))))

(defn run (n input ir)
  (if (zerop n)
      input
    (run (sub1 n) (new-wl input ir) ir)))

(prove-lemma trace-plus (rewrite)
  (equal (behavior-trace (plus i j) x ir)
         (append (behavior-trace i x ir)
                 (behavior-trace j (run i x ir) ir))))

(defn double-cdr-induction (x y)
  (if (nlistp x)
      t
    (double-cdr-induction (cdr x) (cdr y))))

(prove-lemma append-equality-hack1 (rewrite)
  (implies (and (equal (length x) (length y))
                (equal (append x w) (append y z)))
           (equal (equal w z) t))
  ((induct (double-cdr-induction x y))))

(prove-lemma trace-decomposition (rewrite)
  (implies (lessp i n)
           (equal (behavior-trace n x ir)
                  (append (behavior-trace i x ir)
                          (behavior-trace (difference n i) (run i x ir) ir))))
  ((use (trace-plus (j (difference n i))))))

(disable trace-decomposition)

(prove-lemma trace-decomposition2 (rewrite)
  (implies (not (lessp n i))
           (equal (behavior-trace n x ir)
                  (append (behavior-trace i x ir)
                          (behavior-trace (difference n i) (run i x ir) ir))))
  ((use (trace-plus (j (difference n i))))))

(disable trace-decomposition2)

(defn trace-n-hint (n x input ir)
  (if (zerop n)
      t
    (if (nlistp x)
        t
      (trace-n-hint (sub1 n) (cdr x) (new-wl input ir) ir))) )
```

```
(prove-lemma trace-n-bigger (rewrite)
  (implies (and (equal (behavior-trace n input ir)
                       (append x y))
                (listp y))
           (lessp (length x) n))
  ((induct (trace-n-hint n x input ir))
   (enable trace-decomposition)))

(prove-lemma trace-car (rewrite)
  (implies (and (equal (behavior-trace n x ir) lst)
                (listp lst))
           (equal (equal x (car lst)) t)))

(prove-lemma trace-car2 (rewrite)
  (implies (listp (behavior-trace n x ir))
           (equal (car (behavior-trace n x ir)) x)))

(prove-lemma car-run-match (rewrite)
  (implies (and (equal (behavior-trace n input ir)
                       (append init lst))
                (listp lst))
           (equal (run (length init) input ir)
                  (car lst)))
  ((use (append-equality-hack1
         (x (behavior-trace (length init) input ir))
         (y init)
         (w (behavior-trace (difference n (length init))
                   (run (length init) input ir)
                   ir))
         (z lst))
        (trace-decomposition (x input) (i (length init))))
   (do-not-induct t)))

(prove-lemma finding-a-cycle2 (rewrite)
  ;; let's say that the trace is of the form
  ;; (append init (append (cons val stuff) (cons val rest)))
  (implies (and (equal (behavior-trace n input ir)
                       (append init (append stuff rest)))
                (listp stuff)
                (listp rest)
                (equal (car stuff) (car rest)))
           (cyclep stuff rest))
  ((use (append-equality-hack1
         (x (behavior-trace (length init) input ir)) (y init)
         (w (behavior-trace (difference n (length init))
                            (run (length init) input ir)
                   ir))
         (z (append stuff rest)))
        (trace-decomposition (i (length init)) (x input)))
   (enable trace-decomposition2)))

(defn init-segment (x y)
  (if (nlistp x)
      t
    (if (nlistp y)
        f
      (and (equal (car x) (car y))
           (init-segment (cdr x) (cdr y))))))

(prove-lemma init-segment-append (rewrite)
  (implies (init-segment x y)
           (init-segment x (append y z))))

(prove-lemma init-segment-cyclep (rewrite)
  (implies (and (init-segment x y)
                (listp y))
           (cyclep y x))
  ((induct (cyclep y x))))

(prove-lemma init-segment-reflexive (rewrite)
  (init-segment x x))
```

```
(prove-lemma cyclep-reflexive (rewrite)
  (implies (listp x)
           (cyclep x x)))

(prove-lemma cyclep-append-nlistp (rewrite)
  (implies (nlistp z)
           (equal (cyclep x (append y z))
                  (cyclep x y))))

(prove-lemma cycle-append (rewrite)
  (implies (cyclep x y)
           (cyclep x (append x y)))
  ((induct (cyclep x y))))

(prove-lemma finding-a-cycle3 (rewrite)
  ;; This lemma permits me to consider the trace from
  ;; the first repeating element, not merely from the
  ;; second.
  (implies (and (equal (behavior-trace n input ir)
                       (append init (append stuff rest)))
                (listp stuff)
                (listp rest)
                (equal (car stuff) (car rest)))
           (cyclep stuff (append stuff rest)))
  ((use (finding-a-cycle2))))


;;; ************************************************************
(defn repeats (trace)
  ;; This merely says that there is an element
  ;; that appears twice in the trace.
  (if (nlistp trace)
      f
    (if (member (car trace) (cdr trace))
        t
      (repeats (cdr trace)))))

(defn find-repeating-element (trace)
  ;; Find the first repeating element.
  (if (nlistp trace)
      f
    (if (member (car trace) (cdr trace))
        (car trace)
      (find-repeating-element (cdr trace)))))

(defn discard-to (elem trace)
  ;; Throw away elements until the first occurrence
  ;; of elem.
  (if (nlistp trace)
      nil
    (if (equal (car trace) elem)
        trace
      (discard-to elem (cdr trace)))))

(defn up-to (elem trace)
  ;; List the elements of the trace up-to
  ;; an occurrence of elem.
  (if (nlistp trace)
      nil
    (if (equal (car trace) elem)
        nil
      (cons (car trace)
            (up-to elem (cdr trace))))))

(defn find-cycle (trace)
  ;; Notice that this is the same as the later get-cycle
  (let ((x (find-repeating-element trace)))
    (cons x (up-to x (cdr (discard-to x trace))))))

(prove-lemma up-to-discard-to (rewrite)
  (implies (member x lst)
           (equal (append (up-to x lst)
                          (discard-to x lst))
                  lst)))
```

```
(prove-lemma repeats-member (rewrite)
  (implies (repeats trace)
           (member (find-repeating-element trace) trace)))

(prove-lemma member-car-discard-to (rewrite)
   (implies (member x lst)
            (and (listp (discard-to x lst))
                 (equal (car (discard-to x lst)) x))))

(defn repeating-element (x trace)
  (and (member x trace)
       (member x (cdr (discard-to x trace)))))

(defn find-cycle2 (x trace)
  ;; This is like find-cycle but the repeating
  ;; element is specified rather than computed.
  (cons x (up-to x (cdr (discard-to x trace)))))

(defn desired-levelp (x)
  (in-intervalp x (cons 30 70)))

(defn all-in-desired-levelp (lst)
  (if (nlistp lst)
      t
    (and (desired-levelp (car lst))
         (all-in-desired-levelp (cdr lst)))))

(defn trace-ok (trace n)
  (let ((nthcdr (nthcdr n trace)))
    (and (repeats nthcdr)
         (all-in-desired-levelp nthcdr))))

(prove-lemma subset-preserves-all-in-desired-levelp (rewrite)
  (implies (and (all-in-desired-levelp lst)
                (subset lst1 lst))
           (all-in-desired-levelp lst1))
  ((disable desired-levelp)))

(prove-lemma traces-match-longer-trace (rewrite)
    (implies (and (traces-match trace trace2)
                  (not (lessp (length trace2) (length trace)))
                  (repeats trace))
             (repeats trace2)))

(prove-lemma traces-match-longer-trace-nthcdr (rewrite)
    (implies (and (traces-match trace trace2)
                  (not (lessp (length trace2) (length trace)))
                  (repeats (nthcdr m trace)))
             (repeats (nthcdr m trace2))))

(prove-lemma repeats-longer-trace-nthcdr (rewrite)
  (let ((trace (behavior-trace n wl ir))
        (trace2 (behavior-trace p wl ir)))
    (implies (and (repeats (nthcdr m trace))
                  (not (lessp p n)))
             (repeats (nthcdr m trace2))))
  ((use (traces-match-longer-trace-nthcdr
         (trace (behavior-trace n wl ir))
         (trace2 (behavior-trace p wl ir))))))

(prove-lemma repeating-element-member (rewrite)
  (implies (repeats trace)
           (and (member (find-repeating-element trace) trace)
                (member (find-repeating-element trace) (cdr trace)))))

(prove-lemma traces-match-cdr (rewrite)
  (implies (and (listp x)
                (listp y)
                (traces-match x y))
           (traces-match (cdr x) (cdr y))))
```

```
(prove-lemma repeating-element-member2 (rewrite)
  (implies (repeats trace)
           (member (find-repeating-element trace)
                   (cdr (discard-to (find-repeating-element trace)
                                    trace)))))

(prove-lemma trace-match-up-to-match (rewrite)
  (implies (and (traces-match trace trace2)
                (not (lessp (length trace2)
                            (length trace)))
                (member x trace))
           (equal (up-to x trace2)
                  (up-to x trace))))

(prove-lemma length-zero-append (rewrite)
  (implies (zerop (length x))
           (equal (append x y) y)))

(prove-lemma nthcdr-append (rewrite)
  (implies (equal (fix m) (length x))
           (equal (nthcdr m (append x y)) y)))

(prove-lemma behavior-trace-listp (rewrite)
  (equal (listp (behavior-trace n wl ir))
         (not (zerop n))))

(prove-lemma cons-car-crock (rewrite)
  (implies (and (equal (car y) x)
                (listp y))
           (equal (cons x (cdr y)) y)))

(prove-lemma append-cons (rewrite)
  (equal (append (cons x y) z)
         (cons x (append y z))))

(prove-lemma repeating-element-repeats (rewrite)
   (implies (repeating-element x trace)
            (member x (cdr trace))))

(defn cyclep-hint (z x wl cycle ir)
    (if (zerop x)
        t
      (cyclep-hint (sub1 z) (sub1 x)
                   (new-wl wl ir)
                   (append (cdr cycle) (list wl))
                   ir)))

(prove-lemma equal-length-zero-nlistp (rewrite)
  (equal (equal (length x) 0)
         (nlistp x)))

(prove-lemma equal-length-0-init-segment (rewrite)
  (implies (equal (length x) 0)
           (init-segment x y)))

(prove-lemma traces-match-append (rewrite)
  (implies (traces-match (append x y) z)
           (traces-match x z)))

(prove-lemma cycle-match (rewrite)
  (implies (cyclep cycle trace)
           (traces-match cycle trace)))

(defn after-cycle (trace)
  (let ((x (find-repeating-element trace)))
    (discard-to x (cdr (discard-to x trace)))))

(defn before-cycle (trace)
  (let ((x (find-repeating-element trace)))
    (up-to x trace)))

(disable find-cycle)
(disable after-cycle)
(disable before-cycle)
```

```
(prove-lemma trace-decomposition3 (rewrite)
  (implies (repeats trace)
           (equal (append (before-cycle trace)
                          (append (find-cycle trace)
                                  (after-cycle trace)))
                  trace))
  ((enable find-cycle after-cycle before-cycle)))

(disable trace-decomposition3)
(prove-lemma car-find-cycle-after-cycle (rewrite)
  (implies (repeats trace)
           (equal (car (after-cycle trace))
                  (car (find-cycle trace))))
  ((enable find-cycle after-cycle)))

(prove-lemma longer-trace-preserves-member (rewrite)
   (implies (and (member z y)
                 (traces-match x y)
                 (not (lessp (length x) (length y))))
            (member z x)))

(disable init-segment-cyclep)

(prove-lemma longer-trace-preserves-member2 (rewrite)
  (implies (and (member u (behavior-trace z wl ir))
                (not (lessp x z)))
           (member u (behavior-trace x wl ir))))

(prove-lemma member-trace-member-cycle (rewrite)
  (implies (and (member x trace)
                (cyclep cycle trace))
           (member x cycle)))

(prove-lemma car-member-cyclep (rewrite)
  (let ((trace (behavior-trace z wl ir)))
    (implies (and (equal (before-cycle trace) nil)
                  (repeats trace))
             (equal (member x trace)
                    (member x (find-cycle trace)))))
  ((instructions
    (add-abbreviation trace (behavior-trace z wl ir))
    promote
    (use-lemma trace-decomposition3 ((trace trace)))
    (demote 3) (dive 1) s-prop top promote (dive 1 2) =
    (dive 1) = up x up (rewrite member-append) up s-prop split
    (contradict 4)
    (rewrite member-trace-member-cycle
             (($trace (append (find-cycle trace)
                              (after-cycle trace)))))
    (demote 4) (dive 1) s top promote (s lemmas)
    (rewrite finding-a-cycle3
             (($init nil) ($n z) ($input wl) ($ir ir)))
    prove prove prove)))

(prove-lemma car-member-cyclep2 (rewrite)
  (let ((trace (behavior-trace n wl ir)))
    (implies (member wl (behavior-trace (sub1 n) (new-wl wl ir) ir))
             (equal (member x trace)
                    (member x (cons wl (up-to wl (cdr trace)))))))
  ((instructions
    promote (dive 1) (rewrite car-member-cyclep) top
    (s-prop find-cycle) prove (prove (enable before-cycle)) prove)))

(prove-lemma member-up-to-member (rewrite)
  (implies (member x (up-to y trace))
           (member x trace))
  ((disable cycle-match)))
```

```
(prove-lemma longer-trace-preserves-member3-hack (rewrite)
  (implies
   (and (not (equal n 0))
        (numberp n)
        (not (repeats (behavior-trace (sub1 m)
                                      (new-wl wl ir)
                                      ir)))
        (member u
                (behavior-trace (sub1 n)
                                (new-wl wl ir)
                                ir))
        (not (equal m 0))
        (numberp m)
        (repeats (cons wl
                       (behavior-trace (sub1 m)
                                       (new-wl wl ir)
                                       ir)))
        (not (lessp (sub1 n) (sub1 m)))
        (not (equal u wl)))
   (member u
           (behavior-trace (sub1 m)
                           (new-wl wl ir)
                           ir)))
  ((instructions (use-lemma car-member-cyclep2 ((x u)))
                 promote bash promote
                 (rewrite member-up-to-member
                          (($y wl)))
                 (demote 1)
                 (dive 1 2)
                 (= *
                    (up-to wl
                           (behavior-trace (sub1 m)
                                           (new-wl wl ir)
                                           ir))
                    0)
                 top s
                 (dive 1)
                 (rewrite trace-match-up-to-match
                          (($trace (behavior-trace (sub1 m)
                                                   (new-wl wl ir)
                                                   ir))))
                 top s
                 (rewrite behavior-traces-match)
                 prove)))

(disable longer-trace-preserves-member3-hack)

(prove-lemma longer-trace-preserves-member3-hack-2 (rewrite)
  (implies (and (not (repeats (behavior-trace x (new-wl wl ir) ir)))
                (member u (behavior-trace z (new-wl wl ir) ir))
                (member wl (behavior-trace x (new-wl wl ir) ir))
                (not (lessp z x))
                (not (equal u wl)))
           (member u (behavior-trace x (new-wl wl ir) ir)))
  ((use (longer-trace-preserves-member3-hack (n (add1 z)) (m (add1 x))))))

(prove-lemma longer-trace-preserves-member3 (rewrite)
  ;; If wl is a member of a longer trace and a shorter trace
  ;; repeats, then wl is a member of the shorter trace too.
  (implies (and (member u (behavior-trace n wl ir))
                (repeats (behavior-trace m wl ir))
                (not (lessp n m)))
           (member u (behavior-trace m wl ir))))

(disable longer-trace-preserves-member3-hack-2)


;; We have proved a property of the form:
;;    (implies (member u x) (member u y))
;; but really want the form (subset x y), which
;; should follow.  The badguy function is simply
;; an intermediate step along the way.
```

```
(defn badguy (x y)
  (if (listp x)
      (if (member (car x) y)
          (badguy (cdr x) y)
        (car x))
    0))

(prove-lemma subset-suff ()
  (implies (implies (member (badguy x y) x)
                    (member (badguy x y) y))
           (subset x y)))

(prove-lemma longer-trace-subset (rewrite)
  (implies (and (repeats (behavior-trace m wl ir))
                (not (lessp n m)))
           (subset (behavior-trace n wl ir)
                   (behavior-trace m wl ir)))
  ((use (subset-suff
         (x (behavior-trace n wl ir))
         (y (behavior-trace m wl ir))))
   (disable behavior-trace repeats)))

(disable longer-trace-subset)
(prove-lemma all-in-desired-level-longer-trace (rewrite)
  (implies
   (and (not (lessp z x))
        (repeats (behavior-trace x wl ir))
        (all-in-desired-levelp (behavior-trace x wl ir)))
   (all-in-desired-levelp (behavior-trace z wl ir)))
  ((instructions promote
                 (rewrite subset-preserves-all-in-desired-levelp
                          (($lst (behavior-trace x wl ir))))
                 (rewrite longer-trace-subset))))

(defn nthcdr-hint (x y m)
  (if (zerop m)
      t
    (nthcdr-hint (cdr x) (cdr y) (sub1 m))))

(prove-lemma all-in-range (rewrite)
  (let ((trace (behavior-trace n wl ir))
        (trace2 (behavior-trace p wl ir)))
    (implies (and (lessp m n)
                  (not (lessp p n))
                  (trace-ok trace m))
             (trace-ok trace2 m)))
  ((instructions
    promote (demote 3) (s-prop trace-ok) promote split
    (rewrite repeats-longer-trace-nthcdr) (dive 1 2)
    (rewrite trace-decomposition2 (($i m)))
    up (rewrite nthcdr-append) top (demote 3 4)
    (dive 1 1 1 2)
    (rewrite trace-decomposition2 (($i m)))
    up (rewrite nthcdr-append) top (dive 1 2 1 2)
    (rewrite trace-decomposition2 (($i m)))
    up (rewrite nthcdr-append) top promote
    (rewrite all-in-desired-level-longer-trace
             (($x (difference n m))))
    prove prove prove prove prove prove prove)))



(prove-lemma controller-stability-0-100 (rewrite)
  (let ((trace (behavior-trace 30 wl wir)))
    (implies (and (in-intervalp wl (cons 0 100))
                  (in-intervalp wir (cons 10 45)))
             (trace-ok trace 5))))
```

## References

**1.** W.R. Bevier. "Kit and the Short Stack". *Journal of Automated Reasoning 5*, 4 (December 1989), 519-530.

**2.** W.R. Bevier, W.A. Hunt, Jr., J S. Moore, W.D. Young. "An Approach to Systems Verification". *Journal of Automated Reasoning 5*, 4 (December 1989), 411-428.

**3.** W.R. Bevier, W.D. Young. The Proof of Correctness of a Fault-Tolerant Circuit Design. Proceedings of the Second International Working Conference on Dependable Computing for Critical Applications, February, 1991, pp. 107-114.

**4.** R.S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, Inc., ACM Monograph Series, Boston, 1979.

**5.** R. S. Boyer, M. W. Green and J S. Moore. The Use of a Formal Simulator to Verify a Simple Real Time Control Program. In *Beauty Is Our Business*, Springer-Verlag, 1990, pp. 55-64.

**6.** R.S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

**7.** Bishop C. Brock, W.A. Hunt, Jr., and W.D. Young. Introduction to a Formally Defined Hardware Description Language. Proceedings of the IFIP Conference on Theorem Provers in Circuit Design, 1992, pp. 3-36.

**8.** D.M. Goldschlag. "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover". *IEEE Transactions on Software Engineering 16* (September 1990).

**9.** B. Kosko. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence.* Prentice-Hall, Englewood Cliffs, N.J., 1992.

**10.** C.C. Lee. "Fuzzy Logic in Control Systems: Fuzzy Logic Controller-Part I)". *IEEE Transactions on Systems, Man, and Cybernetics 20*, 2 (March-April 1990), .

**11.** C. Lengauer, C.H. Huang. A Mechanically Certified Theorem about Optimal Concurrency of Sorting Networks. Proceedings 13th ACM POPL, 1986, pp. 307-317.

**12.** E.H. Mamdani. "Application of Fuzzy Logic to Approximate Reasoning Using Linguistic Synthesis". *IEEE Transactions on Computers C-26*, 12 (December 1977), 1182-1191.

**13.** J S. Moore. "A Mechanically Verified Language Implementation". *Journal of Automated Reasoning 5*, 4 (December 1989), 493-518.

**14.** M.H. Smith. Parallel Dynamic Switching of Reasoning Methods in a Fuzzy System. Proceedings Fuzz-IEEE, 1993.

**15.** G.L. Steele, Jr. *Common LISP: The Language.* Digital Press, 1984.

**16.** M. Togai and H. Watanabe. "Expert System on a Chip: An Engine for Realtime Approximate Reasoning". *IEEE Expert 1*, 3 (1986).

**17.** W.D. Young. "A Mechanically Verified Code Generator". *Journal of Automated Reasoning 5*, 4 (December 1989), 493-518.

## Table of Contents

List of Figures