THE USE OF A FORMAL SIMULATOR TO VERIFY A SIMPLE REAL TIME CONTROL PROGRAM

Robert S. Boyer
Milton W. Green
J Strother Moore

The work reported here was performed while the authors were in the Computer Science Laboratory, SRI International, Menlo Park, California 94025. Present addresses for the authors are: Boyer, Computer Sciences Department, University of Texas, Austin, Texas 78712; Green, 440 Sherwood Way, Menlo Park 94025; Moore, Computational Logic, Inc., Suite 290, 1717 W. 6th St., Austin, Texas 78703.

This research was supported in part by NASA Contract NAS1-15528, NSF Grant MCS-7904081, and ONR Contract N00014-75-C-0816.

Abstract

We present an initial and elementary investigation of the formal specification and mechanical verification of programs that interact with environments. We describe a formal, mechanically produced proof that a simple, real time control program keeps a vehicle on a straightline course in a variable crosswind. To formalize the specification we define a mathematical function which models the interaction of the program and its environment. We then state and prove two theorems about this function: the simulated vehicle never gets farther than three units away from the intended course and homes to the course if the wind ever remains steady for at least four sampling intervals.

Key Phrases: autopilot, formal specification, mechanical theorem-proving, modeling, program verification, real time control, simulation.

1. Background

Formal computer program verification is a research area in computer science aimed at aiding the production of reliable hardware and software. Formal verification is based on the observation that the properties of a computer program are subject to mathematical proof.

1.1. Program Verification

Consider, for example, the following FORTRAN program for computing integer square roots using a special case of Newton's method[1]

```
      INTEGER FUNCTION ISQRT(I)
      IF ((I .LT. 0)) STOP
      IF ((I .GT. 1)) GOTO 100
      ISQRT = I
      RETURN
100   ISQRT = (I / 2)
200   IF (((I / ISQRT) .GE. ISQRT)) RETURN
      ISQRT = ((ISQRT + (I / ISQRT)) / 2)
      GOTO 200
      END
```

It is possible to prove, mathematically, that the program satisfies the following (informally stated) specification:

> If the program is executed on a machine implementing ANSI FORTRAN 66 or 77 [13, 2], and the input to the program is a nonnegative integer representable on the host machine, then the program terminates, causes no arithmetic overflow or other run time error, and the output is the largest integer whose square is less than or equal to the input.

Such program proofs are generally constructed in two steps. In the first step, the code and its mathematical specifications are transformed into a set of formulas to be proved. In the second step the formulas are proved using the usual laws of logic, algebra, number theory, etc. For an

---

[1] V. Kahan, of U.C. Berkeley, reports that the algorithm was in fact advocated by Heron of Alexandria before 400 A.D.

introduction to program verification, see [9, 10, 11, 1].

Because the mathematics involved in program verification is often tedious and elementary, mechanical program verification systems have been developed. One such system is described in [6]. That system handles a subset of ANSI FORTRAN 66 and 77 and has verified the above mentioned square root program [7], among others.

To admit mechanical proof, the specifications must be written in a completely formal notation. For example, in the square root example the specification of the program's output is:

$$j^2 \leq i < (j+1)^2 \;\&\; 0 \leq j,$$

where it is understood that i refers to the value of the FORTRAN variable I on input to ISQRT and j refers to the value returned by ISQRT.

1.2. Boebert's Challenge

The square root program is a good example of a programming task in which the specification "obviously" captures the intent of the designer. At issue is whether some algorithm satisfies the specification. However, for some programming tasks it is difficult to find mathematical specifications that obviously capture the designer's intention. Real time control programs are an especially important example of such tasks.

To spur the interest of the program verification research community to consider such specification problems, a version of the following problem was proposed by Earl Boebert.[2] Consider the task of steering a vehicle down a straightline course in a crosswind that varies with time. Let the desired course be down the x-axis of a Cartesian plane (i.e, towards increasing values of x). Suppose the vehicle carries a sensor that, in each sampling interval of time, reads either +1, 0, or -1, according to whether the vehicle is to the left of the course (y>0), on the course (y=0), or to the right of the course (y<0). Suppose also that the vehicle has some actuator that can be used to change the y-component of its velocity under the control of some program reading the sensor. Problem: state formally what it means to keep the vehicle on course and, for some particular control program, prove mechanically that the program satisfies its high level specification.

Observe that the problem necessarily involves a specification of the environment with which the program interacts. Furthermore, unlike the square root example, what is desired is not merely a description of a single input/output interchange between the environment and the program but rather the effects of repeated interchanges over time.

In this paper we describe one solution to Boebert's challenge. Our method involves writing a simulator for the system in formal logic. We present our formal simulator after explaining informally the model and control program we will use.

---

[2]Honeywell Systems and Research Center, 2600 Ridgway Parkway, Minneapolis, Minnesota 55413

2. The Informal Model

The mechanized logic into which we cast the model provides the integers and other discrete mathematical objects but does not provide the rationals or reals.[3] Thus, we will measure all quantities, e.g., time, wind speed, vehicle position, etc., in unspecified integral units.

We ignore the x-axis and concentrate entirely on the y-axis. For example, we do not consider the x-component of the vehicle's velocity and we ignore any x-component of the wind velocity. Thus, our model more accurately represents a one-dimensional control problem, such as maintaining constant temperature in an environment where the outside temperature varies, or maintaining constant speed, as in an automobile's "cruise control."

We measure the wind speed, w, in terms of the number of units in the y-direction the wind would blow a passive vehicle in one sampling interval. We assume that from one sampling interval to the next w can change by at most one unit. Some such assumption is required since no control mechanism can compensate for an external agent capable of exerting arbitrarily large instantaneous forces. Thus, we assume that the wind speed at time t+1 is the speed at time t plus some increment, dw, that is either -1, 0, or 1.

```
w(t+1) = w(t) + dw(t+1)
```

where

```
dw(t+1) = -1, 0, or 1.
```

We permit the wind to build up to arbitrarily high velocities.

At each sampling interval the control program may increment or decrement the y-component of its velocity (e.g., by turning a rudder or firing a thruster). We let v be the accumulated speed in the y-direction measured as the number of units the vehicle would move in one sampling interval if there were no wind. We make no assumption limiting how fast v may be changed by the control program; our illustrative program changes v by at most ± 5 each sampling interval. We permit v to become arbitrarily large.

The y-coordinate of the vehicle at time t+1 is thus its y-coordinate at time t, plus the accumulated v at time t, plus the displacement due to the wind at time t+1:

```
y(t+1) = y(t) + v(t) + w(t+1).
```

The sensor reading at any time is the sign of y, sgn(y). The control program changes v at each sampling interval as a function of the current sensor reading (and perhaps previous readings). Our illustrative control program is a function of the current reading and the previously obtained reading:

```
v(t+1) = v(t) + deltav(sen1,sen2)
```

where

---

[3] This is not a limitation of mechanized logic in general. Several existing mechanical theorem-provers, e.g., those of Bledsoe's school [4, 3], and the MAXSYMA symbolic manipulation system [12], provide analytic capability.

```
sen1 = sgn(y(t+1))

sen2 = sgn(y(t)),
```

and deltav is the mathematical function specifying the output of the control program.

3. The Control Program

It is instructive to consider first the control program with the following specification:

```
deltav(sen1,sen2) = -sen1
```

A steadily increasing wind can blow the vehicle arbitrarily far away from the x-axis. Furthermore, should the wind ever become constant, the vehicle begins to oscillate around the x-axis. See Figure 1.

The control program we consider includes a damping term that also causes the vehicle to resist more strongly any initial push away from the x-axis.

```
deltav(sen1,sen2) = -sen1 + 2(sen2-sen1).
```

See Figure 2 for an illustration of the behavior of the vehicle under this program.

The following trivial FORTRAN program implements this specification in the following sense. If SEN1 is the current sensor reading, sen1, and the value of the global variable SEN2 is the previous sensor reading, sen2, and sen1 and sen2 are both legal sensor readings, then at the conclusion of the subroutine, the global ANS is set to deltav(sen1,sen2) and the global SEN2 is set to sen1.

```
SUBROUTINE DELTAV(SEN1)
INTEGER SEN1, SEN2, ANS
COMMON /DVBLK/SEN2, ANS
ANS = ((2 * SEN2) - (3 * SEN1))
SEN2 = SEN1
RETURN
END
```

Proving that the program satisfies its specification is, of course, trivial. At issue is whether the vehicle stays on course.

By observing the behavior of the simulated vehicle under several arbitrarily chosen wind histories we made two conjectures about the behavior of the vehicle:

1. No matter how the wind behaves (within the constraints of the model), the vehicle never strays farther than 3 units away from the x-axis.
2. If the wind ever becomes constant for at least 4 sampling intervals, the vehicle returns to the x-axis and stays there as long as the wind remains constant.

How can we state such specifications in a form that makes them amenable to mechanical proof?

4. Formalizing the Model

To state the conjectures formally we must formalize the model of the control program and its environment. We will define this model as a function in the same mechanized mathematical logic used by the FORTRAN verification system [6]. The logic and a mechanical theorem-prover for it are completely described in [5].

The syntax of the logic is akin to that of Church's lambda-calculus. If f is a function in the logic and e1 and e2 are two expressions in the logic, then we write (f e1 e2) to denote the value of f on the two arguments e1 and e2. The more traditional equivalent notation is f(e1,e2). For example, suppose ZPLUS is defined as the usual integer addition function. Then (ZPLUS X Y) is how we write X+Y. Thus, (ZPLUS 3 -10) = -7.[4]

Our formal model is expressed as a recursive function that takes two arguments, a description of the behavior of the wind over some time period and the initial state of the system. The value of the function is the final state of the system after the vehicle has traveled through the given wind under the direction of the control program. Thus, the recursive function may be thought of as a simulation of the model.

Formally, we let states be triples, <w,y,v>, containing the current wind speed, y-position of the vehicle, and accumulated v. The function STATE, of three arguments, is axiomatically defined to return such a triple, and the functions W, Y, and V are defined to return the respective components of such a triple. Thus, the expression (STATE 63 -2 -61) denotes a state in which the wind speed is 63, the y-position of the vehicle is -2, and the accumulated v is -61.

```
(W (STATE 63 -2 -61)) = 63
(Y (STATE 63 -2 -61)) = -2
(V (STATE 63 -2 -61)) = -61
```

The function NEXT.STATE is defined to return as its value the next state, given the change in the wind and the current state. The formal definition of NEXT.STATE is:

```
Definition.
(NEXT.STATE DW STATE)
   =
(STATE (ZPLUS (W STATE) DW)
       (ZPLUS (Y STATE) (V STATE) (W STATE) DW)
       (ZPLUS (V STATE)
              (DELTAV (SGN (ZPLUS (Y STATE)
                                  (V STATE)
                                  (W STATE)
                                  DW))
                      (SGN (Y STATE))))).
```

The definition of next state follows immediately from our equations for w(t+1), y(t+1) and v(t+1). The function DELTAV is formally defined as was deltav in our informal model.

---

[4]This choice of notation is convenient because most symbols used in program specification are user-defined and do not have commonly accepted names or symbols. Furthermore, the uniformity of the syntax makes mechanical manipulation easier.

The behavior of the wind over n sampling intervals is represented as a sequence of length n. Each element of the sequence is either -1, 0, or 1 and indicates how the wind changes between sampling intervals. Formally, a sequence is either the empty sequence, NIL, or is an ordered pair <hd,tl>, where hd is the first element of the sequence and tl is a sequence containing the remaining elements. Such pairs are returned by the function CONS of two arguments. The functions HD and TL return the respective components of a nonempty sequence, and the function EMPTYP returns true or false according to whether its argument is an empty sequence.

In general we are not interested in wind behaviors other than those permitted by our model. Thus, we define a function that recognizes when an arbitrary sequence consists entirely of -1's, 0's, and 1's.

```
Definition.
(ARBITRARY.WIND LST)
   =
(IF (EMPTY LST)
    T
    (AND (OR (EQUAL (HD LST) -1)
             (EQUAL (HD LST) 0)
             (EQUAL (HD LST) 1))
         (ARBITRARY.WIND (TL LST)))).
```

(ARBITRARY.WIND LST) returns true or false according to whether every element of LST is either -1, 0, or 1. The definition is recursive. The empty sequence has the property. A nonempty sequence has the property provided that (a) the HD of the sequence is -1, 0, or 1, and (b) the TL of the sequence (recursively) has the property.

The recursive function FINAL.STATE takes a description of the wind and an initial state and returns the final state:

```
Definition.
(FINAL.STATE L STATE)
   =
(IF (EMPTY L)
    STATE
    (FINAL.STATE (TL L)
                 (NEXT.STATE (HD L) STATE))).
```

Note that FINAL.STATE is recursively defined and may be thought of as simulating the state changes induced by each change in the wind.

We can now state formally the two properties conjectured earlier.

```
Theorem. VEHICLE.STAYS.WITHIN.3.OF.COURSE:
(IMPLIES (AND (ARBITRARY.WIND LST)
              (EQUAL STATE
                     (FINAL.STATE LST
                                  (STATE 0 0 0))))
         (AND (ZLESSEQP -3 (Y STATE))
              (ZLESSEQP (Y STATE) 3)))
```

This formula may be read as follows. If LST is an arbitrary wind history and STATE is the state of the system after the vehicle has traveled through that wind starting from the initial state <0,0,0>, then the y-coordinate of STATE is between -3 and 3. Put another way, regardless of how the wind behaves, the vehicle is never farther than 3 from the x-axis.

A formal statement of the second conjecture is:

```
Theorem. VEHICLE.GETS.ON.COURSE.IN.STEADY.WIND:
(IMPLIES (AND (ARBITRARY.WIND LST1)
              (STEADY.WIND LST2)
              (ZGREATEREQP (LENGTH LST2) 4)
              (EQUAL STATE
                     (FINAL.STATE (APPEND LST1 LST2)
                                  (STATE 0 0 0))))
         (EQUAL (Y STATE) 0))
```

The function STEADY.WIND recognizes sequences of 0's. The function APPEND is defined to concatenate two sequences. The formula may be read as follows. Suppose LST1 is an arbitrary wind history. Suppose LST2 is a history of 0's at least 4 sampling intervals long. Note that the concatenation of the two histories describes an arbitrary initial wind that eventually becomes constant for at least 4 sampling intervals. Let STATE be the state of the system after the vehicle has traveled through the concatenation of those two wind histories. Then the y-position of the vehicle in that final STATE is 0.

5. Proving the Conjectures

The foregoing conjectures can be proved mathematically. Indeed, they have been proved by the mechanical theorem-prover described in [5]. The key to the proof is that the state space of the vehicle can be partitioned into a small finite number of classes. In particular, any state <w,y,v> reachable under the model starting from <0,0,0> can be put into one of the following classes according to y and w+v:

y	w+v
-3	1
-2	1 or 2
-1	2 or 3
0	-1, 0 or 1
1	-2 or -3
2	-1 or -2
3	-1

The automatic theorem-prover is incapable of discovering this fact for itself. Instead, the human user of the theorem-prover may suggest it by defining the function (GOOD.STATEP STATE) to return true or false according to whether STATE is in one of the 13 classes above, and then commanding the theorem-prover to prove the following key lemma:

```
(IMPLIES (AND (GOOD.STATEP STATE)
              (OR (EQUAL DW -1)
                  (EQUAL DW 0)
                  (EQUAL DW +1)))
         (GOOD.STATEP (NEXT.STATE DW STATE))).
```

This theorem establishes that if the current state of the vehicle is one of the "good states" and the wind changes in an acceptable fashion then the next state is a good state. After proving this lemma (by considering the cases and using algebraic simplification) the theorem-prover can establish by induction on the number of sampling intervals that the final state of the vehicle is a good state. From that conclusion it is immediate that the y-position of the vehicle is within ± 3 of the x-axis.

The proof of the second theorem is similar. The vehicle is in a good state after LST1 has been processed. But if the vehicle is in a good state and the wind remains steady for four sampling intervals, it is easy to show by cases and algebraic simplification that the vehicle returns to the x-axis with w+v=0. But in this case, it stays on the x-axis as long as w stays constant.

6. Comments on the Model

We have proved that the simulated vehicle stays on course under each of the infinite number of different wind histories to which it might be subjected under the model.

Just as the user of a square root or sorting subroutine must look at the specifications to determine whether the subroutine is suitable for his application, so too should the user of this control program. In particular, it is up to the user to determine whether the restrictions on the wind behavior and the model of the environment are sufficiently realistic for his application.

Here are a few of the more obvious oversimplifications:

- Real sensors sometimes give spurious readings due to vibration or other forms of disturbance. The program makes no allowance for such noise.
- No consideration is given to motion or forces in the x- or z-directions. Furthermore, no consideration is given to the orientation of the vehicle with respect to its preferred direction of travel.
- The model of the physics of the vehicle is too simple. The use of discrete measurement is unsatisfying but perhaps justifiable under suitable assumptions about scale. But many physical aspects of real control situations have been ignored: inertia, reaction times of the actuators, response time of the vehicle, maximum permitted g-forces.

Allowance for noise in the sensors can be handled by existing program verification technology. For example, if one provides redundant sensors and employs a signal select algorithm based on software majority voting, DELTAV can be rewritten to use an algorithm such as that verified in [8] to compute the majority sensor reading (if any). The proof that the vehicle stays on course can then be carried over directly if one is willing to assume that at each sampling interval a majority of the sensors agree.

However, the other two unrealistic aspects of our problem are more difficult to handle. While it is easy to define more sophisticated formal simulators it may well be practically impossible to prove interesting properties mechanically. Certainly the proof paradigm used here, depending as it did on the existence of a small partitioning of the state space, will not suffice for more sophisticated models.

7. Conclusion

We have illustrated how a formal simulator can be used to specify in a machine readable form the high level intention of a simple real time control program. We have also shown how such a program has been mechanically proved to satisfy its specifications.

Simulation programs are used today to test a variety of applications programs. Among the applications that come to mind are real time control, scheduling, and page fault handling in operating systems. Such simulators suffer the inaccuracy introduced by finite precision arithmetic and resources and in addition offer only the testing of the applications program on a finite number of situations.

Formal simulators are mathematical functions. They need not be realizable on machines and thus need not suffer resource limitations. In addition, formal simulators theoretically permit mechanical analysis of the behavior of the system in an infinite number of possible situations.

References

1. R. B. Anderson. *Proving Programs Correct.* John Wiley & Sons, New York, New York, 1979.

2. American National Standards Institute, Inc. American National Standard Programming Language FORTRAN. Tech. Rept. ANSI X3.9-1978, American National Standards Institute, Inc., 1430 Broadway, N.Y. 10018, April, 1978.

3. A. M. Ballantyne and W. W. Bledsoe. Automatic Proofs of Theorems in Analysis using Non-Standard Techniques. Tech. Rept. ATP-23, Department of Mathematics, University of Texas at Austin, July, 1975.

4. W. Bledsoe, R. Boyer, and W. Henneman. "Computer Proofs of Limit Theorems". *Artificial Intelligence 3* (1972), 27-60.

5. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

6. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

7. R. S. Boyer and J S. Moore. The Mechanical Verification of a FORTRAN Square Root Program. SRI International, 1981.

8. R. S. Boyer and J S. Moore. MJRTY - A Fast Majority Vote Algorithm. Technical Report ICSCA-CMP-32, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

9. R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, American Mathematical Society, Providence, Rhode Island, 1967, pp. 19-32.

10. J. C. King. *A Program Verifier*. Ph.D. Th., Carnegie-Mellon University, 1969.

11. Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill Book Company, New York, New York, 1974.

12. J. Moses. Algebraic Simplification: A Guide for the Perplexed. 2nd Symposium on Symbolic and Algebraic Manipulation, ACM, 1971.

13. United States of America Standards Institute. USA Standard FORTRAN. Tech. Rept. USAS X3.9-1966, United States of America Standards Institute, 10 East 40th Street, New York, New York 10016, 1966.

Table of Contents