What Can Formal Methods Offer to Digital Flight Control Systems Design?

Donald I. Good
Computational Logic, Inc.

Abstract

Formal methods research is beginning to produce methods which will enable mathematical modeling of the physical behavior of digital hardware and software systems. The development of these methods directly supports the NASA mission of increasing the scope and effectiveness of flight system modeling capabilities.

The conventional, continuous mathematics that is used extensively in modeling flight systems is not adequate for accurate modeling of digital systems. Therefore, the current practice of digital flight control system design has not had the benefits of extensive mathematical modeling which are common in other parts of flight system engineering.

Formal methods research is showing that by using discrete mathematics, very accurate modeling of digital systems is possible. These discrete modeling methods are still in an embryonic stage. But when they are fully developed, they will bring the traditional benefits of modeling to digital hardware and software design. Sound reasoning about accurate mathematical models of flight control systems can be an important part of reducing the risks of unsafe flight control.
What Can Formal Methods Offer to Digital Flight Control Systems Design?

Donald I. Good

Computational Logic, Inc.
1717 West Sixth, Suite 290
Austin, Texas 78703

512-322-9951

good@cli.com
"Formal Methods" Enable Mathematical Modeling of Digital Systems (Hardware and Software)

Why Model?

For either design of a new system or operation of an old one, modeling provides...

Benefits: early error detection
- Saves time
- Saves money
- Saves operational disruption
- Saves operational mishaps

Risks: model misrepresents system
- Inaccurate
- Incomplete

Kinds of models: physical, analog, schematic, mathematical.

Why a Mathematical Model?

- High abstraction
- High precision
- Simulate by manipulating symbols
- Represent large classes of system states
- Use mathematical deduction

Get a lot of system simulation for a little symbol manipulation.
Operational Safety

Operating a system safely requires

- accurate predictions

of how it will behave.

Accurate predictions can be obtained from

- sound deductions about
- accurate mathematical models

of system behavior.
A Classic Model

Free Fall Distance:

\[ f(b, t) = \frac{[g(b) \times t^2]}{2} \]

\[ g(b) = \begin{cases} 32 & \text{if } b = \text{"earth"} \\ \text{...} & \text{else if } b = \text{"moon"} \end{cases} \]

t is time (sec)

\[ f(b,t) \] is distance (ft)

Simulation:

\[ f(\text{"earth"}, .7) = \frac{[32 \times .7^2]}{2} \]

\[ = 16 \times .49 \]

\[ = 7.84 \text{ ft} \]
Power of Mathematical Deduction

Suppose $0 \leq t_0 \leq t_1$.

\[ t \in [t_0..t_1] \]

\[ f("earth", t) \in \frac{(32 \times [t_0..t_1]^{**2})}{2} \]

\[ f("earth", t) \in 16 \times [t_0..t_1]^{**2} \]

\[ f("earth", t) \in 16 \times [t_0**2..t_1**2] \]

**(** is monotonic**)

Physical simulation of this result is impossible because \([t_0..t_1]\) contains an infinite number of values.
Validating a Model

- Ultimately, the **accuracy** of a model of a physical system must be **validated** by testing it against measured, observed behavior of the actual physical system.

- One cannot construct a **mathematical proof** that a model is an **accurate** representation of a **physical** system.

- Typically, one iterates through a process of
  - **stating** a mathematical model
  - **testing** it against physical observations
  - **adjusting** the model
Hardware Model Observables

A hardware system is composed of physical switches.


Next page.
Use **Discrete** Mathematics to Model Hardware

- **Switches** by binary digits

- **Operation** by recursive functions

<table>
<thead>
<tr>
<th>s0</th>
<th>0 1 1 0 0 0 0 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>1 0 1 0 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td>s2</td>
<td>1 1 1 0 0 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>
An MC68020 Machine Model

MC68020(s,n) =
    if haltp(s) or n=0
        then s
    else MC68020(NEXT(s), n-1)

NEXT(s) =
    if evenp(pc(s))
        then if pc_readp(mem(s), pc(s))
            then EXECUTE(FETCH(pc(s), s),
                         update_pc(s,...))
        else halt(s, pc_signal)
    else halt(s, pc_odd_signal)

EXECUTE(ins,s) =
    ... [50 pages for 90% user ins.] ...

Provides a mathematically precise and consistent machine language reference manual.

The VIPER Machine

A 32-bit microprocessor "whose functions are totally predictable."

- Accumulator
- 2 index registers
- Program counter
- Comparison register
- 16 instructions


A VIPER Machine Model

NEXT(ram, p, a, x, y, b, stop) =
   if  stop
   then (ram, p, a, x, y, b, stop)
   else (noinc \ illegaladdr) \/
            if  (illegalcl \ illegalsp)
                \ (illegalonp \ illegalwr)
              then (ram, newp, a, x, y, b, T)
            else ...

where

ram - a memory of 32-bit words
p - 20-bit program counter
a - 32-bit accumulator
x, y - 32-bit index registers
b - 1 bit compare result register
stop - stop flag
The FM8502 Machine

A 32-bit microprocessor.

- 2 address architecture
- 4 addressing modes
- 8 general purpose registers
- \(2^{19}\) 20-bit instructions


An FM8502 Machine Model

FM8502(ms,mn) =
    if not(listp(mn))
    then ms
    else FM8502(NEXT(ms),
                  rest(mn))

NEXT(ms) =
    list(next_memory (ms),
         next_register_file(ms),
         next_carry_flag  (ms),
         next_overflow_flag(ms),
         next_zero_flag   (ms),
         next_negative_flag(ms) )

... [about 10 pages] ...
An FM8502
Register Transfer Model

GATES(gs, gn) =
    if not(listp(gn))
    then gs
    else GATES(COMB_LOGIC(gs, car(gn)),
                cdr(gn))

COMB_LOGIC(gs, gn) =
... [on bit operators, e.g., b_xor] ...

where

gs       - [regs, flags, mem, int-regs]
regs     - 8 32-bit vectors
flags    - 4 Booleans
mem      - $2^{32}$ 32-bit vectors
int-regs - 32-bit vectors for internal
            registers, flags, latches
Connecting the Models

\[ \text{Theorem: } H(ms, mn) \rightarrow \]
\[ \text{fm8502}(ms, mn) = \]
\[ U(\text{gates}(D(ms), K_g(ms, mn, md))) \]

Under the conditions $H$,

- the fm8502 model is just as accurate as gates
- but with some details suppressed by $U$. 
Software Model Observables

Programming languages provide a wide variety of ways of describing them, but the observables are still switches, and so are programs!
Models of Programmed Machines

- A machine is programmed by setting the switches which it will interpret as instructions during its operation. (Before stored-program machines, this process was called "setting up" the machine.)

  \[
  \begin{array}{c|ccc|cccc}
  \hline
  \text{prog} & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
  \hline
  \end{array}
  \]

- These switches are the program. They control the subsequent operation of the machine.

- A computer program is a physical control mechanism.

- The bit string "011000" is a mathematical description of the control mechanism.
A Model of a Programmed Machine

A model of machine $M$ operating on initial state $s_0$ for $k(s_0)$ steps under the control of the program described by $p_0$ is given by

$$M(s_0, k(s_0))$$

where

$s_0$ - a machine state such that $\text{prog}(s_0) = p_0$

$\text{prog}(s)$ - a function that extracts the program description from $s$

Operating Requirements

A model of a machine programmed to satisfy an operating requirement $R(s_0, s_k)$ is given by

$$R(s_0, M(s_0, k(s_0)))$$
A Program Description, p0

[752 16-bit words]
The Kit Separation Kernel

- Uses a modified FM8501 (ms, mn) machine
- Interrupts for timer and I/O
- Process management
  - fixed number of processes
  - process scheduling (round robin)
  - process communication (message passing)
  - response to error conditions
- Device management for character I/O to asynchronous devices
- Memory management uses hardware protection

Kit Operating Requirement, R
The CLInc Stack

\[ o \longrightarrow u\text{Gypsy}(yx, yp, yd, yn) \longrightarrow o \]
\[ \downarrow \]
\[ \text{Compile} \quad \text{Young} \quad p\text{-display} \]
\[ \downarrow \]
\[ v \]
\[ o------ p\text{iton}(ps, pn) \longrightarrow o \]
\[ \downarrow \]
\[ \text{Link-assemble} \quad \text{Moore} \quad m\text{-display} \]
\[ \downarrow \]
\[ v \]
\[ o------ f\text{m8502}(ms, mn) \longrightarrow o \]
\[ \downarrow \]
\[ \text{Reify} \quad \text{Hunt} \quad g\text{-display} \]
\[ \downarrow \]
\[ v \]
\[ o------ g\text{ates}(gs, gn) \longrightarrow o \]

The Piton Language

The Piton language has

- execute-only program space
- read/write global arrays
- recursive subroutine calls
- formal parameters
- user-visible stack
- stack-based instructions
- flow-of-control instructions.

The cross assembler produces an FM8502 binary core image.
The Micro Gypsy Language

The Micro Gypsy subset of Gypsy has

- types integer, boolean, character
- one dimensional arrays
- procedure calls with pass by reference parameters
- sequential control structures if, loop,
- condition handling signal..when.

The compiler produces Piton.
The Stack Theorem

Theorem: $H'(yx, yp, yd, yn) \rightarrow$

$$uGypsy(yx, yp, yd, yn) =$$

$$U'(gates(D'(yx, yp, yd),$$

$$Kg'(yx, yp, yd, yn, md)))$$

Proof: Mechanically checked.

Under the conditions $H'$,

- the $uGypsy$ model is just as accurate as $gates$
- but with many details suppressed by $U'$.

Boyer-Moore Logic


A Hierarchy of Models of a Programmed Machine

\[ R(yx_0, yp_0, yd_0, ydk) \]
\[ uGypsy(yx_0, yp_0, yd_0, yk(yx_0, yp_0, yd_0)) \]
\[ piton(ps_0, pk(ps_0)) \]
\[ fm8502(ms_0, mk(ms_0)) \]
\[ gates(gs_0, gk(gs_0)) \]

Corresponding to these is a hierarchy of program descriptions....
procedure mult(var ans: fm8502_int;
i, j: fm8502_int) =

begin
ENTRY j ge 0;
EXIT ans = NTIMES(i, j);
pending;
end;

type fm8502_int =
    integer[-(2**31)..(2**31)-1];

{A Simple Problem Domain Theory}

function NTIMES(x, y: integer) : integer =
begin
exit (assume result =
    if y = 0 then 0
    else if y = 1 then x
    else x + NTIMES(x, y-1)
    fi fi);
Gypsy Program Description

procedure mult(var ans:fm8502_int;
               i,j:fm8502_int) =
begin
ENTRY  j ge 0;
EXIT   ans = NTIMES(i,j);
   var k:fm8502_int := 0;
    k := j;
   ans := 0;
loop
   ASSERT j ge 0 & k in [0..j]
      & ans = NTIMES(i,j-k);
   if k le 0 then leave end;
   ans := ans + i;
    k := k - 1;
end;
end;
Piton Program Description

(MG-MULT
  (K ZERO ONE B ANS I J) ;formals
NIL ;locals
(PUSH-LOCAL ANS) ;ans := 0;
(PUSH-CONSTANT (INT 0))
(CALL MG-SIMPLE-CONSTANT-ASSIGNMENT)
(PUSH-LOCAL K) ;k := j;
(PUSH-LOCAL J)
(CALL MG-SIMPLE-VARIABLE-ASSIGNMENT)
(DL L-1 NIL (NO-OP)) ;loop
(PUSH-LOCAL B) ; b := k le 0
(PUSH-LOCAL K)
(PUSH-LOCAL ZERO)
(CALL MG-INTEGRAL-LE)
(PUSH-LOCAL B) ; if b then leave
(FETCH-TEMP-STK)
(TEST-BOOL-AND-JUMP FALSE L-3)
(PUSH-CONSTANT (NAT 0))
(POP-GLOBAL C-C)
(JUMP L-2)
(JUMP L-4)
(DL L-3 NIL (NO-OP))
(DL L-4 NIL (NO-OP))
(PUSH-LOCAL ANS) ; ans := ans + i;
(PUSH-LOCAL ANS)
(PUSH-LOCAL I)
(CALL MG-INTEGRAL-ADD)
(PUSH-GLOBAL C-C)
... [14 more support routines] ...
FM8502 Program Description

... [10 more pages] ...}
Mathematical Requirements

• Unambiguous: Requirements have a well-defined interpretation that tells exactly what they do say.

• Analyzable: Do the requirements say the "right" thing?

  \[ R(x, y) \rightarrow \text{good\_thing}(x, y) \]

• Consistency: Requirements contain no contradictions.

• Enable modeling a program component before building it (and thereby save the time and cost of designing a poor program.)

To get these benefits, the requirements notation must have a rigorous mathematical foundation (semantics).
Design >> Requirements

- There is more to designing a digital system than just stating and refining mathematical requirements.
- One must still construct a program for some machine.
- Mathematical models of commonly used languages and machines are still very scarce.
Summary

For either design of a new system or operation of an old one, mathematical modeling of digital flight control systems offers

Benefits: early error detection
- Saves time
- Saves money
- Saves operational disruption
- Saves operational mishaps

Risks: model misrepresents system
- Inaccurate
- Incomplete
Conventional Non-Wisdom

Use "formal methods" (mathematical modeling)

- only after a system is built to certify it
- only before a system is built to design it
- to guarantee perfect system behavior
- to eliminate the need for testing