可信的国家软件资源
共享与协同生产环境

Formal Method is Working
The Formal Development of VMK
- An Operating System Kernel

Ming-Yuan Zhu

CoreTek Systems

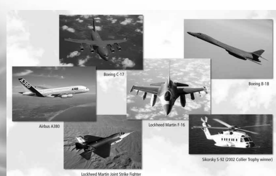
June 2009

嵌入式系统联谊会
http://www.esbf.org.cn

Contents

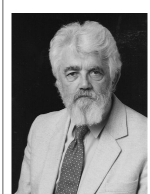
可信的国家软件资源
共享与协同生产环境

Introduction



Is McCarthy's Dream Practical?

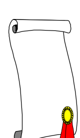
Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program.



John McCarthy,
"A Basis for a Mathematical Theory of Computation,"
1961

Fundamental Questions

❖Fundamental questions in physical science: What is the nature of matter? What is the basis and origin of organic life?
❖Fundamental questions in computer science: What is an algorithm? What can and what cannot be computed? When should an algorithm be considered practically feasible?
❖Fundamental questions in operating systems: What is an OS? How is an OS implemented? What constitute a safe, reliable and efficient OS?



The Correctness Problems of Operating Systems

❖Why software are always incorrect?
❖Why most of commercial OSes are always incorrect?
❖How to make a correct program?
- Debugging?
- Testing?
- Or whatever?

What is Wrong?!

A Classification

- The Market Proven Operating Systems: Those operating systems have been in market for over 10-15 years and have been used in many safety-critical applications. They are therefore tested by the customers for a long time and many errors have been found and fixed. This kind of operating systems include pSOSystem, VxWorks and VRTX. However, this does not mean that these operating systems are safe and reliable.
- The Certificated Operating Systems: Those operating systems have been validated by some internationally recognized organizations through a very systematic verification and validation approach. This kind of operating system includes OSE, Ada and Integrity 178B.
- The Provably Correct Operating Systems: The operating system which has been proven to be correct against a formal specification.

Importance and Feasibility

- An embedded RTOS is so important that it is necessary to prove its correctness.
- An embedded RTOS kernel is so small that it is possible to prove its correctness.

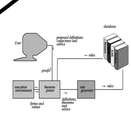
How Do We Know It Works?

- We can test it?
- We can monitor its development process?
- We can prove it!

High Assurance Systems

DO-178B (DO-254) Software/System Assurance Levels

- **Level A: Catastrophic Failure Protection**
- **Level B: Hazardous/Severe Failure Protection**
- **Level C: Major Failure Protection**
- **Level D: Minor Failure Protection**
- **Level E: Minimal Failure Protection**



Common Criteria Evaluation Assurance Levels

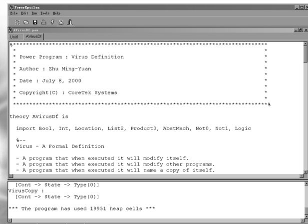
- **EAL 7: Formally Verified Design and Tested**
- **EAL 6: Semi-formally Verified Design and Tested**
- **EAL 5: Semi-formally Designed and Tested**
- **EAL 4: Methodically Designed, Tested and Reviewed**
- **EAL 3: Methodically Tested and Checked**
- **EAL 2: Structurally Tested**
- **EAL 1: Functionally Tested**

Common Criteria Evaluation Levels

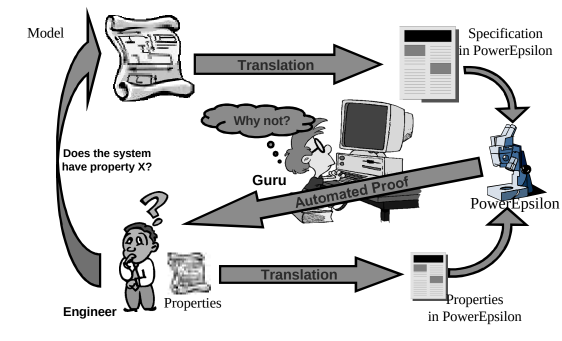
Common Criteria	Requirements	Functional Specification	HLD	LLD	Implementation
EAL 1	Informal	Informal	Informal	Informal	Informal
EAL 2	Informal	Informal	Informal	Informal	Informal
EAL 3	Informal	Informal	Informal	Informal	Informal
EAL 4	Informal	Informal	Informal	Informal	Informal
EAL 5	Formal	Semiformal	Semiformal	Informal	Informal
EAL 6	Formal	Semiformal	Semiformal	Semiformal	Informal
EAL 7	Formal	Formal	Formal	Semiformal	Informal
Verified	Formal	Formal	Formal	Formal	Formal

可信的国家软件资源
共享与协同生产环境

Proof Development System - PowerEpsilon



Theorem Proving Process in PowerEpsilon



PowerEpsilon

- A strongly-typed polymorphic functional programming language based on Martin-Lof's Type Theory and Calculus of Constructions.
- The concept of type universe hierarchies and a scheme for inductive defined types are introduced.
- The system can be used as both a programming language with a very rich set of data structures and a meta-language for formalizing constructive mathematics.
- The system has been implemented using the software development system AUTOSTAR.

Natural Induction Rule

```
dec NatInduction :
  ∀(P : [Nat -> Prop])
   [@(P, OO) ->
    ∀(m : Nat) [@(P, m) -> @(P, @(SS, m))] ->
    ∀(n : Nat) @(P, n)];
```

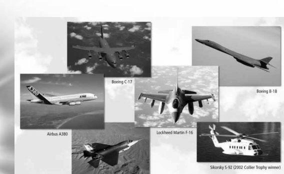
Theory - Leibniz's Equality

```
theory Equal is

  def Equal =
    λ(A : Type(0),
      x : A,
      y : A)
     ∀(P : [A -> Prop])
      [@(P, x) -> @(P, y)]

end;
```

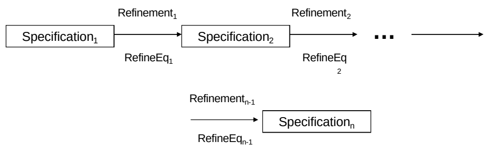
```
dec REFLEX :
  ∀(A : Type(0),
    R : [A -> A -> Prop], x : A)
   @(R, x, x);

dec SYMM :
  ∀(A : Type(0),
    R : [A -> A -> Prop],
    x : A, y : A)
   [@(R, x, y) -> @(R, y, x)];

dec TRANS :
  ∀(A : Type(0),
    R : [A -> A -> Prop],
    x : A, y : A, z : A)
   [@(R, x, y) -> @(R, y, z) -> @(R, x, z)]
```

可信的国家软件资源
共享与协同生产环境

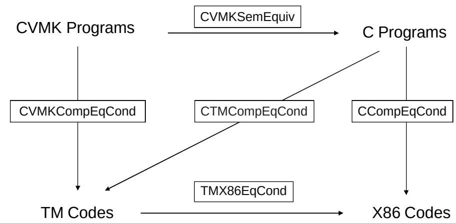
Proof Structure



Specification Development



Proof Structure



Source Languages and Semantics

```
%-- CVMK Source Programs --%

dec CVMKProgram : Prop;

%-- C Source Programs --%

dec CProgram : Prop;

%-- CVMK Semantics --%

dec CVMKState : Prop;

dec CVMKProgSEM : [CVMKProgram -> CVMKState -> CVMKState];

%-- C Semantics --%

dec CState : Prop;

dec CProgSEM : [CProgram -> CState -> CState];
```

CVMK and C Semantic Equivalence

```
dec CVMKStEquiv : [CVMKState -> CState -> Prop];

dec CVMK2CTran : [CVMKProgram -> CProgram];

def CVMKSemEquiv =
  λ(p1 : CVMKProgram, p2 : CProgram)
   let c1 = @(CVMKProgSEM, p1),
       c2 = @(CProgSEM,    p2) in
      ∀(z1 : CVMKState, z2 : CState)
       [@(CVMKStEquiv, z1, z2) ->
        @(CVMKStEquiv, @(c1, z1), @(c2, z2))];
```

Target Machine

```
%-- TM Target Instructions --%
dec TMInstr : Prop;
def TMInstrList = @(List, TMInstr);

%-- TM Target Machines --%
dec TMState : Prop;
```

```
%-- X86 Target Instructions --%
dec X86Instr : Prop;
def X86InstrList = @(List, X86Instr);

%-- X86 Target Machines --%
dec X86State : Prop;
```

Semantics of Target Codes

```
%-- Semantics of TM Target Codes --%
dec TMSem : [TMInstrList -> TMState -> TMState];

%-- Semantics of X86 Target Codes --%
dec X86Sem : [X86InstrList -> X86State -> X86State];
```

```
%-- TM and X86 Target Equivalence --%
dec TMEquiv : [TMState -> X86State -> Prop];
```

The Compilers

```
%-- CVMK to TM Compilers --%

dec CVMKCompiler : [CVMKProgram -> TMInstrList];

%-- C to X86 Compilers --%

dec CCompiler : [CProgram -> X86InstrList];

%-- C to TM Compilers --%

dec C2TMCompiler : [CProgram -> TMInstrList];

%-- TM Loader --%

dec TMLoad : [TMInstrList -> TMState -> TMState];

%-- X86 Loader --%

dec X86Load : [X86InstrList -> X86State -> X86State];
```

Semantic Equivalence of CVMK and TM

```
dec CVMKTMStEquiv : [CVMKState -> TMState -> Prop];

def CVMKCompEqCond =
  λ(p : CVMKProgram, q : TMInstrList)
   let c1 = @(CVMKProgSEM, p),
       c2 = @(TMLoad, q) in
     ∀(z1 : CVMKState, z2 : TMState)
      [@(CVMKTMStEquiv, z1, z2) -> @(CVMKTMStEquiv, @(c1, z1), @(c2, z2))];

def CVMKCompEquiv =
  λ(p : CVMKProgram) @(CVMKCompEqCond, p, @(CVMKCompiler, p));

dec CVMKCompEqThm :
  ∀(p : CVMKProgram)
   ∃(q : TMInstrList) @(CVMKCompEqCond, p, q);

dec CVMKCompEqThm2 :
  ∃(f : [CVMKProgram -> TMInstrList])
   ∀(p : CVMKProgram) @(CVMKCompEqCond, p, @(f, p));
```

Semantic Equivalence of C and TM

```
dec CTMStEquiv : [CState -> TMState -> Prop];

def CTMCompEqCond =
  λ(p : CProgram, q : TMInstrList)
   let c1 = @(CProgSEM, p),
       c2 = @(TMLoad, q) in
     ∀(z1 : CState, z2 : TMState)
      [@(CTMStEquiv, z1, z2) -> @(CTMStEquiv, @(c1, z1), @(c2, z2))];

def CTMCompEquiv = λ(p : CProgram) @(CTMCompEqCond, p, @(C2TMCompiler, p));

dec CTMCompEqThm :
  ∀(p : CProgram) ∃(q : TMInstrList) @(CTMCompEqCond, p, q);

dec CTMCompEqThm2 :
  ∃(f : [CProgram -> TMInstrList])
   ∀(p : CProgram) @(CTMCompEqCond, p, @(f, p));
```

Semantic Equivalence of C and X86

```
dec CX86StEquiv : [CState -> X86State -> Prop];

def CCompEqCond =
  λ(p : CProgram, q : X86InstrList)
   let c1 = @(CProgSEM, p), c2 = @(X86Load, q) in
     ∀(z1 : CState, z2 : X86State)
      [@(CX86StEquiv, z1, z2) ->
       @(CX86StEquiv, @(c1, z1), @(c2, z2))];

def CCompEquiv =
  λ(p : CProgram) @(CCompEqCond, p, @(CCompiler, p));

dec CCompEqThm :
  ∀(p : CProgram)
   ∃(q : X86InstrList) @(CCompEqCond, p, q);
```

Semantic Equivalence of TM and X86

```
dec TM2X86Tran : [TMInstrList -> X86InstrList];

dec TMX86StEquiv : [TMState -> X86State -> Prop];

def TMX86EqCond =
  λ(q : TMInstrList, r : X86InstrList)
   let c1 = @(TMSem, q),
       c2 = @(X86Sem, r) in
     ∀(z1 : TMState, z2 : X86State)
      [@(TMX86StEquiv, z1, z2) ->
       @(TMX86StEquiv, @(c1, z1), @(c2, z2))];
```

Semantic Equivalence of CVMK and X86 Through TM

```
def CVMKPrX86EqCond1 =
  λ(p : CVMKProgram,
    q : TMInstrList,
    r : X86InstrList)
   @(And,
     @(CVMKCompEqCond, p, q),
     @(TMX86EqCond, q, r));

dec CVMKPrX86EqThm1 :
  ∀(p : CVMKProgram)
   ∃(q : TMInstrList, r : X86InstrList)
    @(CVMKPrX86EqCond1, p, q, r);
```

Semantic Equivalence of CVMK and X86 Through C

```
def CVMKPrX86EqCond2 =
  λ(p : CVMKProgram,
    q : CProgram,
    r : X86InstrList)
   @(And,
     @(CVMKSemEquiv, p, q), @(CCompEqCond, q, r));

dec CVMKPrX86EqThm2 :
  ∀(p : CVMKProgram)
   ∃(q : CProgram, r : X86InstrList)
    @(CVMKPrX86EqCond2, p, q, r);
```

Semantic Equivalence of CVMK and X86 Through C and TM

```
def CVMKPrX86EqCond3 =
  λ(p : CVMKProgram,
    q : CProgram,
    t : TMInstrList,
    r : X86InstrList)
   @(And,
     @(CVMKSemEquiv, p, q),
   @(And,
     @(CTMCompEqCond, q, t),
     @(TMX86EqCond, t, r)));

dec CVMKPrX86EqThm3 :
  ∀(p : CVMKProgram)
   ∃(q : CProgram, t : TMInstrList, r : X86InstrList)
    @(CVMKPrX86EqCond3, p, q, t, r)
```

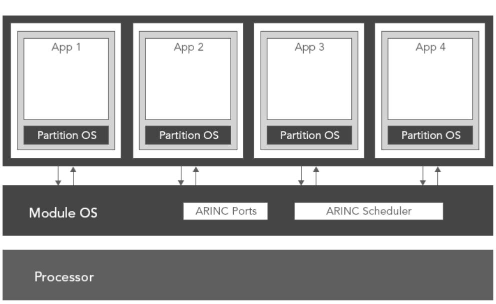
可信的国家软件资源
共享与协同生产环境

Functional Specifications

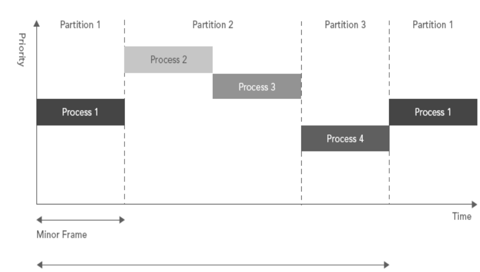
Gold Standard for Partitioning

- A partitioned system should provide **fault containment** equivalent to an idealized system in which each partition is allocated an independent processor and associated peripherals and all inter-partition communications are carried on **dedicated lines**.
  - Partition:
    - Spatial (processor, memory, resources)
    - Temporal (processor cycles)
  - The propagation of fault effects is prevented
  - Communication lines are independent

Partition-Based Architecture

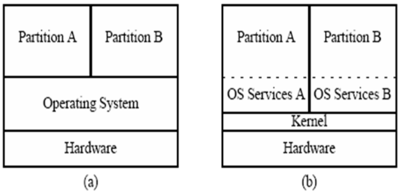


Temporal Partitioning



Levels of Partitioning

Partitions could have their own ‘copies’ of OS services:



Alternative Operating System/Partitioning Designs

Separation Kernel Condition

For a given VMK state z, for any vmcb1 of type VMCB, if vmcb1 is well-defined in z, for any address l, if l is well-defined in z and vmcb1, for any vmcb2 of type VMCB, if vmcb2 is well-defined in z and l is well-defined in z and vmcb2, then vmcb1 and vmcb2 are equal.

```
def SepKernelCond =
  λ(z : VMKState)
   ∀(vmcb1 : VMCB)
    [@(VMCB_In_VMKState, vmcb1, z) ->
     ∀(l : VMKAddress)
      [@(Addr_In_VMCB, l, z, vmcb1) ->
       ∀(vmcb2 : VMCB)
        [@(VMCB_In_VMKState, vmcb2, z) ->
         @(Addr_In_VMCB, l, z, vmcb2) ->
         @(Equal, VMCB, vmcb1, vmcb2)]]];
```

Separation Kernel Theorem

For any VMK state z and vmcb1 of type VMCB, if vmcb1 is well-defined in z, for any address l, if l is well-defined in z and vmcb1, for any vmcb2 of type VMCB, if vmcb2 is well-defined in z and l is well-defined in z and vmcb2, then vmcb1 and vmcb2 are equal.

```
dec SepKernelThm :
  ∀(z : VMKState) @(SepKernelCond, z);
```

The Proof of Separation Kernel Theorem

```
def SepKernelLem =
  λ(z : VMKState, vmcb1 : VMCB, p : @(VMCB_In_VMKState, vmcb1, z))
  λ(l : VMKAddress), q : @(Addr_In_VMCB, l, z, vmcb1), vmcb2 : VMCB)
  λ(q1 : @(VMCB_In_VMKState, vmcb2, z), q2 : @(Addr_In_VMCB, l, z, vmcb2))
   let vmindex1 = @(GetVMCBMemoIndex, vmcb1),
       vmindex2 = @(GetVMCBMemoIndex, vmcb2) in
     let addr_idx = @(GET_VMKADDR_IDX, l),
         addr_loc = @(GET_VMKADDR_LOC, l) in
       let vmcb_idx1 = @(GetVMCBMemoIndex, vmcb1),
           vmcb_idx2 = @(GetVMCBMemoIndex, vmcb2) in
         let P11 = @(Equal, VMKMemoIndex, addr_idx, vmcb_idx1),
             P12 = let nz = @(SET_VMKST_IDX, z, addr_idx) in
                     let v = @(GET, nz, addr_loc) in
                       @(Not, @(Equal, KSval, v, ERR_KSVAL)),
             P21 = @(Equal, VMKMemoIndex, addr_idx, vmcb_idx2),
             P22 = let nz = @(SET_VMKST_IDX, z, addr_idx) in
                     let v = @(GET, nz, addr_loc) in
                       @(Not, @(Equal, KSval, v, ERR_KSVAL)) in
           let p11 = @(PJ1, P11, P12, q), p21 = @(PJ1, P21, P22, q2) in
             let w = @(Symm_Eq, VMKMemoIndex, addr_idx, vmcb_idx1, p11) in
               @(Tran_Eq, VMKMemoIndex, vmcb_idx1, addr_idx, vmcb_idx2, w, p21);
```

The Proof of Separation Kernel Theorem (Cont)

```
def SepKernelThm =
  λ(z : VMKState, vmcb1 : VMCB, p : @(VMCB_In_VMKState, vmcb1, z))
  λ(l : VMKAddress, q : @(Addr_In_VMCB, l, z, vmcb1))
  λ(vmcb2 : VMCB)
  λ(q1 : @(VMCB_In_VMKState, vmcb2, z),
    q2 : @(Addr_In_VMCB, l, z, vmcb2))
   let P = @(Equal, VMCB, vmcb1, vmcb2) in
     let w = @(ExclMidRule, P) in
       @(WHEN,
         P,
         @(Not, P),
         P,
         w,
         λ(u : P) u,
         λ(u : @(Not, P))
          let v1 = @(UniqVMCBIdxThm1, z, vmcb1, vmcb2, p, q1, u),
              v2 =
                 @(SepKernelLem, z, vmcb1, p, l, q, vmcb2, q1, q2) in
           @(v1, v2, P));
```

可信的国家软件资源
共享与协同生产环境

High-Level Design and Abstract Machine

The Challenges

- System programs - especially those involving both interrupts and concurrency - are extremely hard to reason about.
- Mixture of high-level and low-level programming techniques in OS development.
- Most difficult part: modeling of interrupt handling
- Existing program verification techniques can probably handle those high-level concurrent programs, but they have consistently ignored the issues of interrupts thus cannot be used to certify concurrent code in the OS kernel code. Having both explicit interrupts and threads creates the new challenges.

Two Layers of Abstraction

- At the “higher” abstraction level, we have threads that follow the standard concurrent programming model: interrupts are invisible, but the execution of a thread can be preempted by other threads; synchronization operations are treated as primitives.
- Below this layer, we have more subtle “lower-level” code involving both interrupts and concurrency. The implementation of many synchronization primitives and input/output operations requires explicit manipulation of interrupts.

Instructions of VMK-TM

- TM-1
  - Instructions for Computing and Control
- TM-2
  - Instructions for VMK Manager, VMCBs and TTSCBs
  - Instructions for Virtual Interruption Management
  - Instructions for Scheduling
  - Instructions for Location List

Semantics of Context Saving

```
dec SAVE_CONTEXT : [TMState -> TMState];

def SAVE_CONTEXT =
  λ(z : TMState)
   let im = @(GET_TMST_IMEM,   z),
       dm = @(GET_TMST_DMEM,   z),
       rg = @(GET_TMST_TMREG,  z),
       st = @(GET_TMST_TMSREG, z),
       io = @(GET_TMST_TMVECT, z),
       mn = @(GET_TMST_VMKMAN, z) in
     let old_vm_loc = @(GET_VMKMAN_RUNVM, mn) in
       let old_vm = @(VMCB_GET, z, old_vm_loc) in
         let vm_stack  = @(GetVMCBIntContxt, old_vm),
             nvm_stack = @(PUSH, TMReg, rg, vm_stack) in
           let new_vm = @(SetVMCBIntContxt, old_vm, nvm_stack) in
             @(VMCB_SET, z, old_vm_loc, new_vm);

def SV_CONT_SEM =
  λ(z : TMState, r : Nat, s : Nat, t : Nat)
   @(SAVE_CONTEXT, z);
```

Semantics of Context Switching

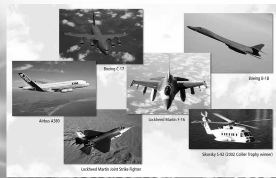
```
dec SWITCH_CONTEXT : [TMState -> TMState];

def SWITCH_CONTEXT =
  λ(z : TMState)
   let im = @(GET_TMST_IMEM,   z), dm = @(GET_TMST_DMEM,   z),
       rg = @(GET_TMST_TMREG,  z), sg = @(GET_TMST_TMSREG, z),
       io = @(GET_TMST_TMVECT, z), mn = @(GET_TMST_VMKMAN, z) in
     let run_vm_loc = @(GET_VMKMAN_RUNVM, mn) in
       let run_vm = @(VMCB_GET, z, run_vm_loc) in
         let memoidx = @(GetVMCBMemoIndex, run_vm),
             cnstack = @(GetVMCBIntContxt, run_vm) in
           let context  = @(TOP, TMReg, ERR_TMREG, cnstack),
               ncnstack = @(POP, TMReg, cnstack) in
             let nrg =
               @(TMREG_ASSIGN, context, SG_REG, @(NAT2B32, memoidx)) in
               let nrun_vm = @(SetVMCBIntContxt, run_vm, ncnstack) in
                 let nz = @(SET_TMST_TMREG, z, nrg) in
                   @(VMCB_SET, z, run_vm_loc, nrun_vm);

def SW_CONT_SEM =
  λ(z : TMState, r : Nat, s : Nat, t : Nat)
   @(SWITCH_CONTEXT, z);
```

Simulation Semantics of TM

```
def TARGET_MACHINE =
  λ(z : TMState)
   let im = @(GET_TMST_IMEM,   z), dm = @(GET_TMST_DMEM,   z),
       rg = @(GET_TMST_TMREG,  z), st = @(GET_TMST_TMSREG, z),
       io = @(GET_TMST_TMVECT, z) in
     let ie = @(TMREG_RETRIEVE, rg, IE_REG), ic = @(TMREG_RETRIEVE, rg, IC_REG),
         im = @(TMREG_RETRIEVE, rg, IM_REG), ih = @(TMREG_RETRIEVE, rg, IH_REG),
         sm = @(TMREG_RETRIEVE, rg, SG_REG) in
       @(GIF_THEN_ELSE, TMState, @(IS_INT_ENABLED, ie),
         @(GIF_THEN_ELSE, TMState, @(IS_INT_FIRED, ic),
           let n = @(GET_1ST_FIRED_INT, ic) in
             @(GIF_THEN_ELSE, TMState, @(GTBit32Word, im, n),
               let z1 = @(FETCH_EXEC_CYCLE, z), z2 = @(EXTERNAL_ENV, z1) in
                 @(TARGET_MACHINE, z2),
               let nie = INT_DISABLE,    nic = @(STBit32Word, ic, n, FF),
                   nsm = @(NAT2B32, OO), nst = @(PUSH, TMReg, rg, st),
                   nrg = @(GetIntHandler, ih, n, z) in
                 let nrg1 = @(TMREG_ASSIGN, nrg,  IE_REG, nie),
                     nrg2 = @(TMREG_ASSIGN, nrg1, IC_REG, nic),
                     nrg3 = @(TMREG_ASSIGN, nrg2, SG_REG, nsm) in
                   let z1 = @(SET_TMST_TMREG,  z,  nrg3), z2 = @(SET_TMST_TMSREG, z1, nst),
                       z3 = @(FETCH_EXEC_CYCLE, z2),      z4 = @(EXTERNAL_ENV, z3) in
                     @(TARGET_MACHINE, z4)),
           let z1 = @(FETCH_EXEC_CYCLE, z), z2 = @(EXTERNAL_ENV, z1) in
             @(TARGET_MACHINE, z2)),
         let z1 = @(FETCH_EXEC_CYCLE, z), z2 = @(EXTERNAL_ENV, z1) in
           @(TARGET_MACHINE, z2));
```

可信的国家软件资源
共享与协同生产环境

Conclusions



Related Works in 2000s

- ❖ Rockwell Collins AAMP7 Separation Kernel Microcode
- ❖ Rockwell Collins/Green Hills Integrity OS Kernel
- ❖ Sun Microsystems JVM

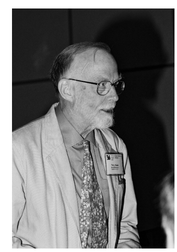
Impossible Dreams of Science

- ❖ Physics: accuracy of measurement
- ❖ Chemistry: purity of materials
- ❖ Biology: rational drug design
- ❖ Computer Science: zero defect programs

- Tony Hoare, 2007

The Dream is Possible!

By combining the work of scientists
who pursue long-term ideals
with the work of engineers
who pursue immediate advantage
to develop a program verifier,
and realise the dream
of zero defect programming.
within the next fifteen years



- Tony Hoare, 2007

Formal Methods Humor

- Mechanical Engineering is like looking for a black cat in a lighted room.
- Chemical Engineering is like looking for a black cat in a dark room.
- Software Engineering is like looking for a black cat in a dark room in which there is no cat.
- Systems Engineering is like looking for a black cat in a dark room in which there is no cat and someone yells, “I got it!”

Questions?

Thank You!