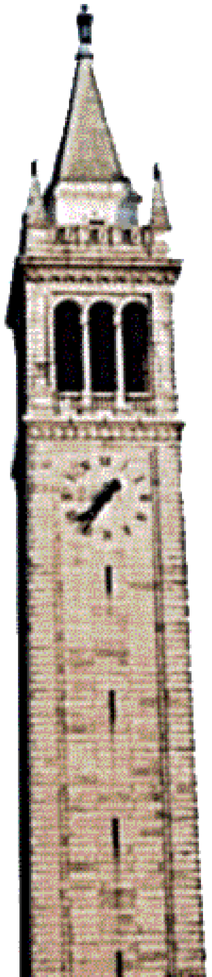


Certifying Real-Time Software is Not Reasonable (Today)

Edward A. Lee

Robert S. Pepper Distinguished Professor
UC Berkeley

Invited Plenary Talk

Software Certification Consortium (SCC) Workshop at the High Confidence Software and Systems (HCSS) Conference

Annapolis, MD

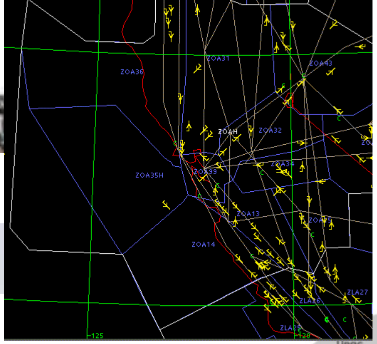
May 6 and 7, 2012.

Key Collaborators on work shown here:

- *David Broman*
- *Steven Edwards*
- *Sungjun Kim*
- *Isaac Liu*
- *Hiren Patel*
- *Jan Reinke*
- *Mike Zimmer*

Cyber-Physical Systems (CPS): *Orchestrating networked computational resources with physical systems*

Avionics

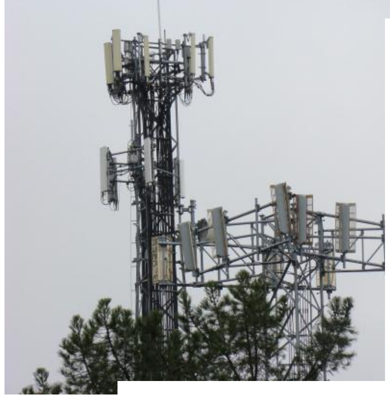


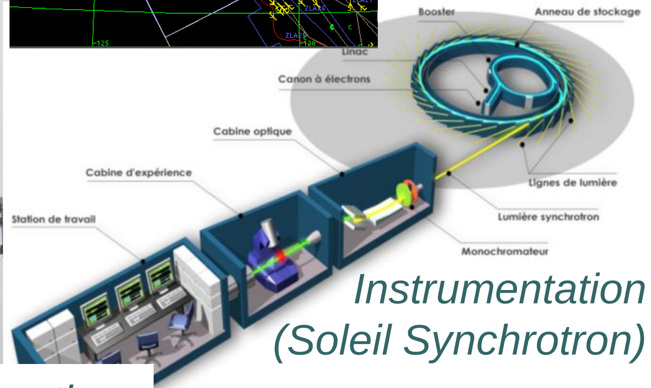
Transportation (Air traffic control at SFO)

Building Systems

Elevator
Switch
Server room
Fire and security monitoring
Electricity

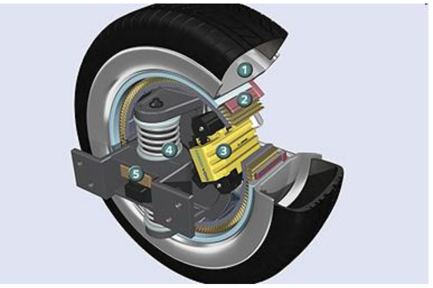
Telecommunications



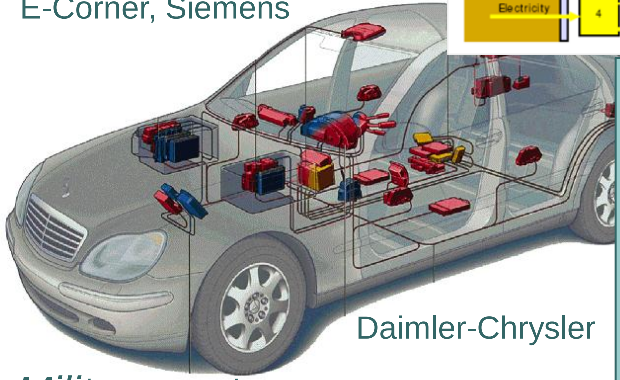


Instrumentation (Soleil Synchrotron)

Automotive



E-Corner, Siemens



Daimler-Chrysler

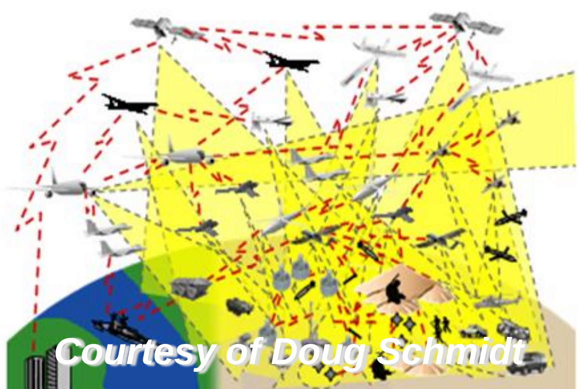
Power generation and distribution

Courtesy of General Electric

Factory automation

Courtesy of Kuka Robotics Corp.

Military systems:



Courtesy of Doug Schmidt

Claim

For CPS, *programs* do not adequately specify *behavior*.

Corollary: Certifying *software* for CPS makes no sense.

A Story

The Boeing 777 was Boeing’s first fly-by-wire aircraft, controlled by software. It is deployed, appears to be reliable, and is succeeding in the marketplace. Therefore, it must be a success. However…

Boeing was forced to purchase and store an advance supply of the microprocessors that will run the software, sufficient to last for the estimated 50 year production run of the aircraft and another many years of maintenance.

Why?

Lesson from this example:

Apparently, the software does not specify the behavior that has been validated and certified!

Unfortunately, this problem is very common, even with less safety-critical, certification-intensive applications. Validation is done on complete system implementations, not on software.

Guidance for Certification: The Koptez Principle

Prof. Dr. Hermann Kopetz

Many properties that we assert about systems (determinism, timeliness, reliability) are in fact not properties of an *implemented* system, but rather properties of a *model* of the system.

We can make definitive statements about *models*, from which we can *infer* properties of system realizations. The validity of these properties depends on *model fidelity*, which is always approximate.

(paraphrased)

Corollary

Certifying models requires clear understanding of model semantics.

You have to know what a model means to assert that it holds certain properties.

Software as-a Model

C program specifying timed behavior.

```
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    SysTickIntRegister(&ISR);
    .. // other init
    timer_count = 2000;
    initTimer();
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
    ... // other code
}
```

Software as-a Model

C program specifying timed behavior.

Within the semantics of C, ISR() is never called in main().

```
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    SysTickIntRegister(&ISR);
    .. // other init
    timer_count = 2000;
    initTimer();
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
    ... // other code
}
```

Software as-a Model

C program specifying timed behavior.

Within the semantics of C, ISR() is never called in main().

Within the semantics of C, how long will this code run?

```
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    SysTickIntRegister(&ISR);
    .. // other init
    timer_count = 2000;
    initTimer();
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
    ... // other code
}
```

Timing is not Part of Software Semantics

Correct execution of a program in C, C#, Java, Haskell, OCaml, etc. has nothing to do with how long it takes to do anything. All our computation and networking abstractions are built on this premise.

Programmers have to step *outside* the programming abstractions to specify timing behavior.

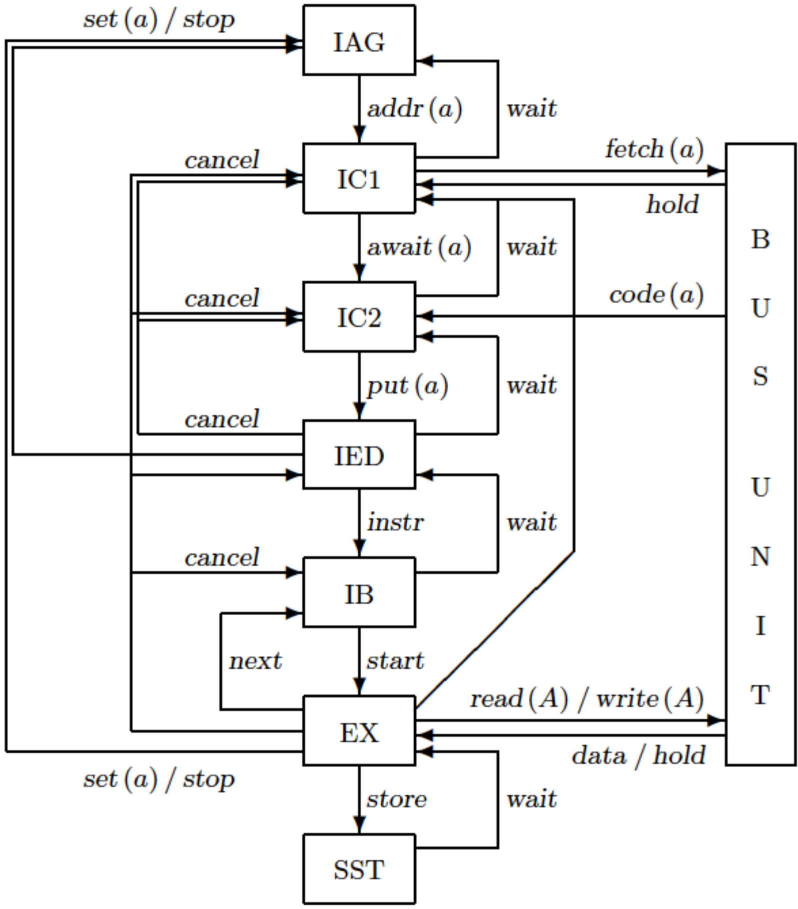
Execution-time analysis, by itself, does not solve the problem!

Analyzing software for timing behavior requires:

- Paths through the program (undecidable)
- Detailed model of microarchitecture
- Detailed model of the memory system
- Complete knowledge of execution context
- Many constraints on preemption/concurrency
- Lots of time and effort

And the result is valid only for that exact hardware and software!

Fundamentally, the programming language and the ISA of the processor have failed to provide adequate abstractions.



Wilhelm, et al. (2008). "The worst-case execution-time problem - overview of methods and survey of tools." ACM TECS 7(3): p1-53.

In contrast, some definitive statements about software are possible

We can safely assert that line 8 does not execute

(In C, we need to separately ensure that no other thread or ISR can overwrite the stack, but in more modern languages, such assurance is provided by construction.)

```
1  void foo(int32_t x) {
2          if (x > 1000) {
3                  x = 1000;
4          }
5          if (x > 0) {
6                  x = x + 1000;
7                  if (x < 0) {
8                          panic();
9                  }
10         }
11 }
```

We can develop absolute confidence in the software, in that only a hardware failure is an excuse.

How about an RTOS?

“Real-time” operating systems (RTOS’s), allow us to specify scheduling priorities, but ultimately rely on **execution time analysis** and **overprovisioning** to provide assurance (and then, only on a particular hardware platform in a particular execution context).

The software, as a model, is not adequate for certification!

We can fix this problem!

First Part of Our Solution: PRET Machines

- **PRE**cision-**T**imed processors = **PRET**
- **P**redictable, **RE**peatable **T**iming = **PRET**
- **P**erformance *with* **RE**peatable **T**iming = **PRET**

```
// Perform the convolution.
for (int i=0; i<10; i++) {
  x[i] = a[i]*b[j-i];
  // Notify listeners.
  notify(x[i]);
}
```

+

= PRET

Computing

With time

Dual Approach

- Rethink the ISA
  - Timing has to be a *correctness* property not a *performance* property.
- Implementation has to allow for multiple realizations and efficient realizations of the ISA
  - Repeatable execution times
  - Repeatable memory access times

Example of one sort of mechanism we would like:

```
tryin (500ms) {
  // Code block
} catch {
    panic();
}
```

```
jmp_buf  buf;

if ( !setjmp(buf) ){
  set_time r1, 500ms
  exception_on_expire r1, 0
  // Code block
  deactivate_exception 0
} else {
    panic();
}

exception_handler_0 () {
     longjmp(buf)
}
```

Pseudocode showing how this might be implemented today. The result is very platform dependent.

If the code block takes longer than 500ms to run, then the panic() procedure will be invoked.

But then we would like to verify that panic() is never invoked!

Extending an ISA with Timing Semantics

[V1] Best effort:

```
set_time r1, 1s
// Code block
delay_until r1
```

[V2] Late miss detection

```
set_time r1, 1s
// Code block
branch_expired r1, <target>
delay_until r1
```

[V3] Immediate miss detection

```
set_time r1, 1s
exception_on_expire r1, 1
// Code block
deactivate_exception 1
delay_until r1
```

[V4] Exact execution:

```
set_time r1, 1s
// Code block
MTFD r1
```

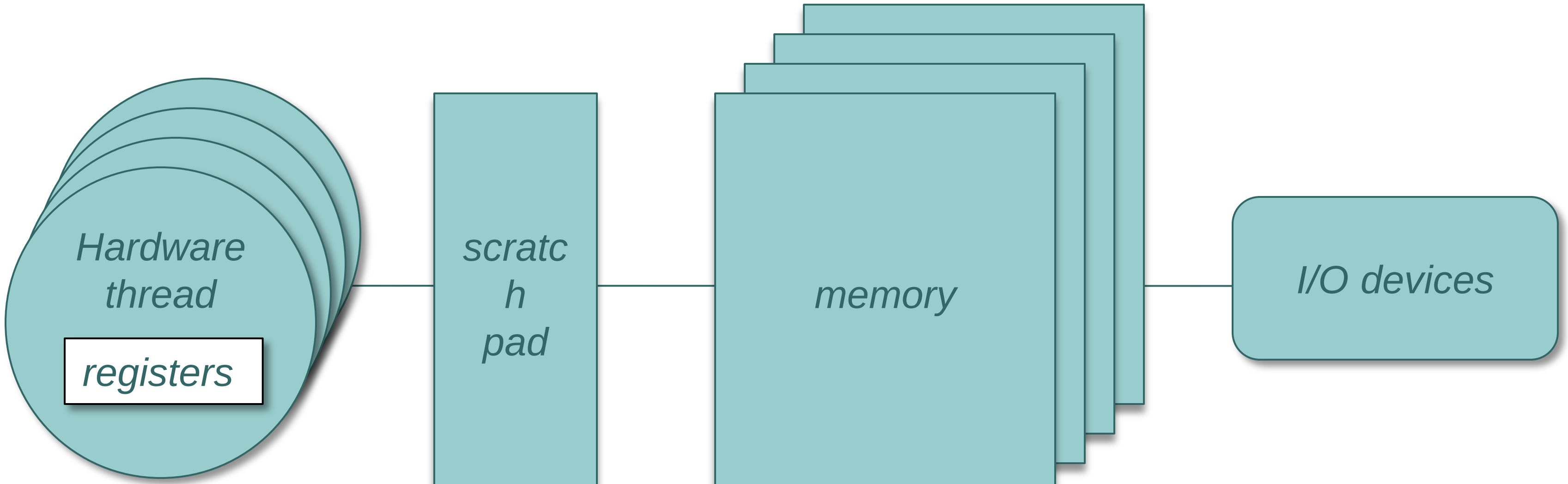
To deliver repeatable timing, we have to rethink the microarchitecture

Challenges:

- Pipelining
- Memory hierarchy
- I/O (DMA, interrupts)
- Power management (clock and voltage scaling)
- On-chip communication
- Resource sharing (e.g. in multicore)

Our Current PRET Architecture

PTArm, a soft core on a Xilinx Virtex 5 and 6 FPGA



Interleaved pipeline with one set of registers per thread

SRAM scratchpad shared among threads

DRAM main memory, separate banks per thread

Performance Cost?

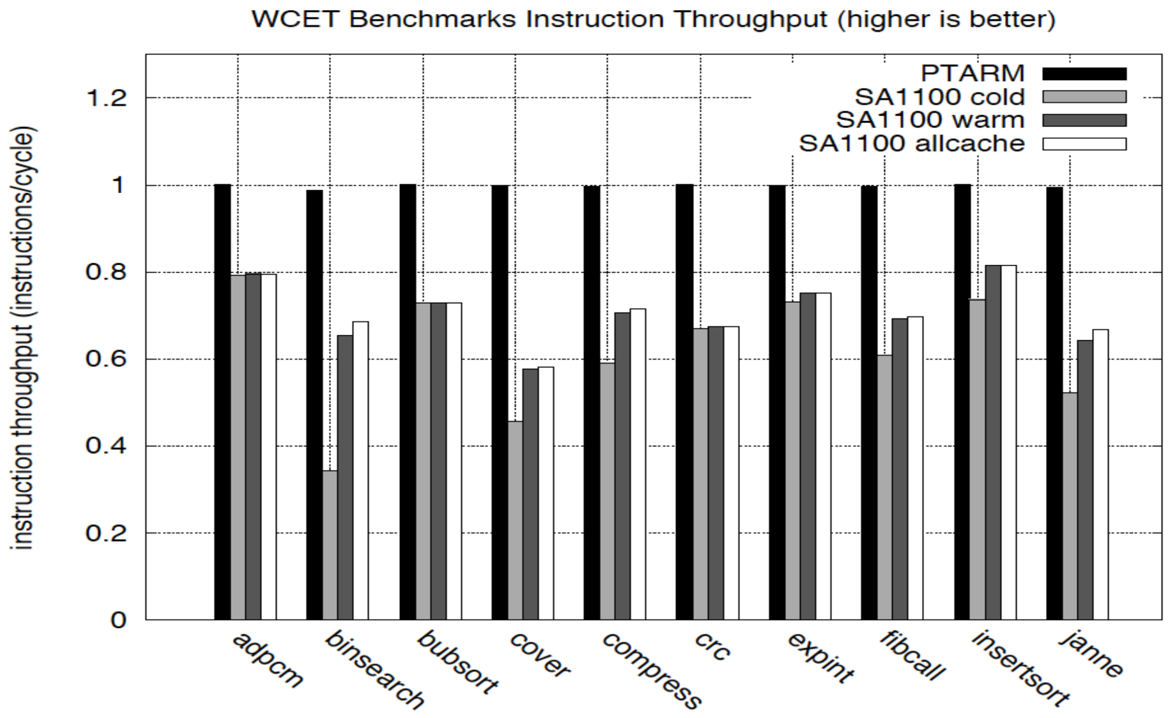
No!

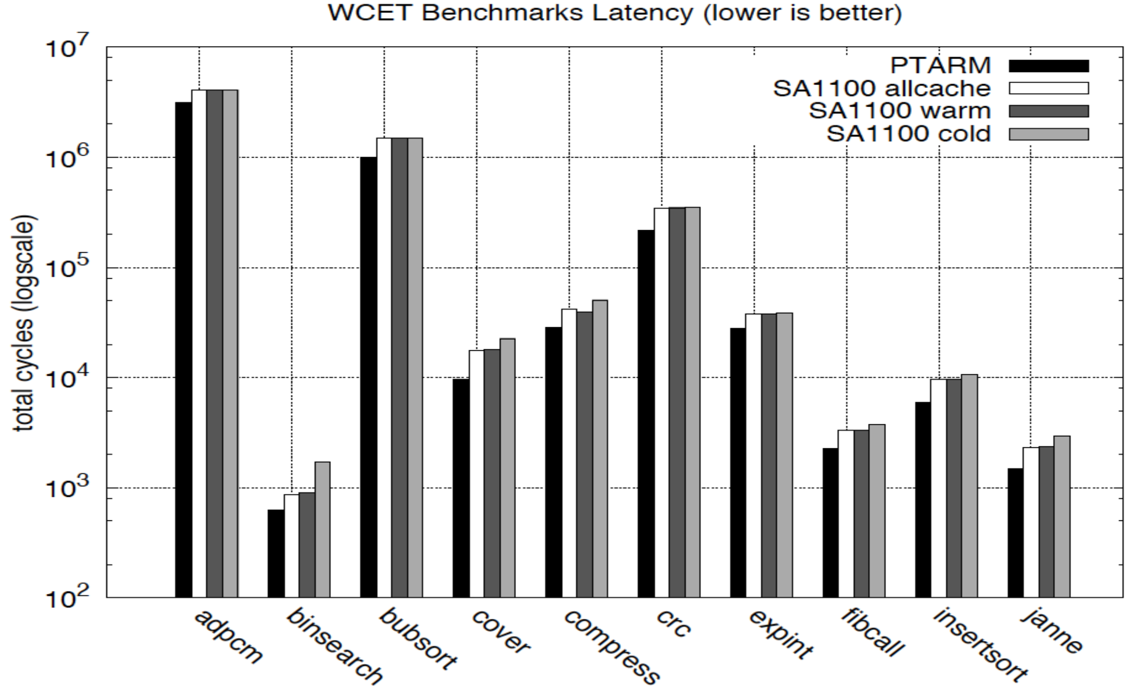
Comparing PTARM against SimIT-ARM simulator (StrongARM 1100) [Qin & Malik] over Malardalen WCET benchmarks [Gustafsson…].

Given enough concurrency, the PTARM beats the StrongARM on every benchmark!

Moreover, our simpler pipeline can probably be clocked faster.

[Isaac Liu, PhD Thesis, May, 2012]





A Key Next Step: Parametric PRET Architectures

```
set_time r1, 1s
// Code block
MTFD r1
```

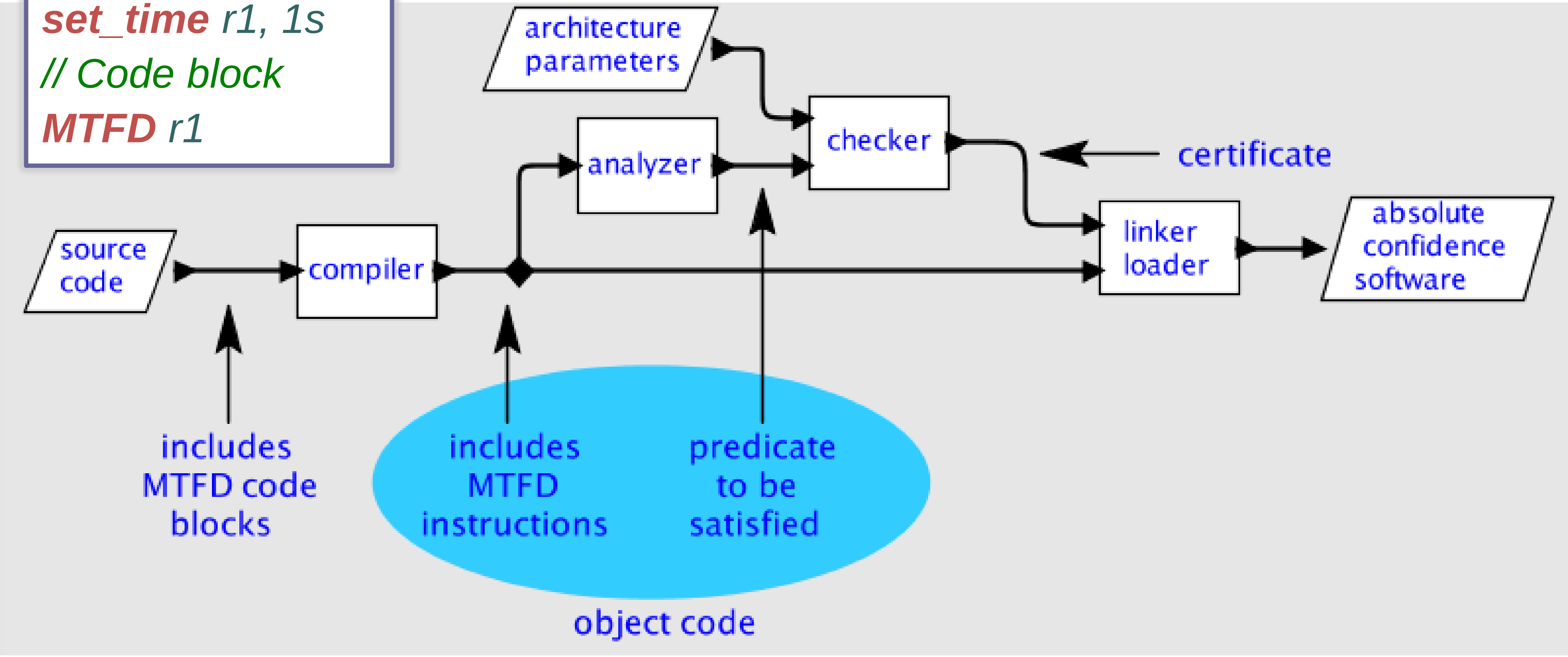
ISA that admits a variety of implementations:

- Variable clock rates and energy profiles
- Variable number of cycles per instruction
- Latency of memory access varying by address
- Varying sizes of memory regions
- …

A given program may meet deadlines on only some realizations of the same parametric PRET ISA.

Realizing the MTFD instruction on a parametric PRET machine

```
set_time r1, 1s
// Code block
MTFD r1
```


architecture parameters
checker
analyzer
certificate
source code
compiler
linker loader
absolute confidence software
includes MTFD code blocks
includes MTFD instructions
predicate to be satisfied
object code

The goal is to make software that will run correctly on a variety of implementations of the ISA, and that correctness can be checked for each implementation.

PRET Publications

http://chess.eecs.berkeley.edu/pret/

- S. Edwards and E. A. Lee, "**The Case for the Precision Timed (PRET) Machine**," in the *Wild and Crazy Ideas* Track of DAC, June 2007.
- B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards and E. A. Lee, "**Predictable programming on a precision timed architecture**," CASES 2008.
- S. Edwards, S. Kim, E. A. Lee, I. Liu, H. Patel and M. Schoeberl, "**A Disruptive Computer Design Idea: Architectures with Repeatable Timing**," ICCD 2009.
- D. Bui, H. Patel, and E. Lee, "**Deploying hard real-time control software on chip-multiprocessors**," RTCSA 2010.
- Bui, E. A. Lee, I. Liu, H. D. Patel and J. Reineke, "**Temporal Isolation on Multiprocessing Architectures**," DAC 2011.
- J. Reineke, I. Liu, H. D. Patel, S. Kim, E. A. Lee, **PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation**, CODES+ISSS, Taiwan, October, 2011.
- S. Bensalem, K. Goossens, C. M. Kirsch, R. Obermaisser, E. A. Lee, J. Sifakis, **Time-Predictable and Composable Architectures for Dependable Embedded Systems**, Tutorial Abstract, EMSOFT, Taiwan, October, 2011

Conclusions

Today, timing behavior is a property only of *realizations* of software systems.

Tomorrow, timing behavior will be a semantic property of *programs* and *models*.

Raffaello Sanzio da Urbino – The Athens School