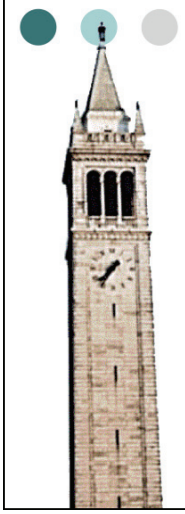


Predictability, Repeatability, and Models for Cyber-Physical Systems

Gage Eads
Marc Geilen
Edward A. Lee <speaker>
Ben Lickly
Isaac Liu
Jan Reineke
Maarten Wiggers

Invited Talk

Foundations and Applications of Component-based Design (WFCD)
ArtistDesign Workshop organized within Embedded Systems Week
Scottsdale, AZ, October 24, 2010

Abstract

To be able to analyze and control the joint dynamics of software, networks, and physical processes, engineers need to be able to analyze and control the temporal behavior of software and networks. Most real-time systems techniques aim to do this by focusing on predicting execution time bounds through program analysis and architectural modeling. In this talk, I attempt to give a rigorous meaning to "prediction" in this context, and argue that what engineers need is not predictability so much as repeatability. The argument relies on the observation that Boolean properties (properties that are either satisfied or not satisfied) are necessarily properties of models and not properties of the physical realizations of systems. Predictability is about the ability to anticipate whether properties will be satisfied, whereas repeatability is about the ability of a physical realization to deliver consistent behavior.

Oft heard dogma

"What we need is predictable timing."

I would like to examine this statement…

Influences

- **Kopetz**:
  Properties are held by *models* not by *realizations*.
- **Henzinger**:
  Boolean properties vs. continuity
- **Lee & Sangiovanni-Vincentelli**:
  Tagged signal model
- **Wilhelm**:
  The gold standard in predictability

This is a work in progress!!!

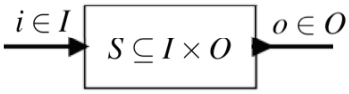
A snippet of our ongoing discussion:

What does *every* quantify over? Does a program contain its inputs or should the property hold for all possible inputs (in that case, I would argue that such a property is also immediately *predictable*)? (Marc: The alternative definition might solve this?) (Isaac: Right, this is tricky because we can define the *property* we want repeatable to be either including the inputs or not. Although i would presume that the latter (holding true for all possible situations) is more useful, but harder to achieve (or is that possible if we don't constrain the inputs? What about an atomic bomb drop?)) (Jan: Yeah, asking for repeatability under all possible inputs is asking for too much to be practically feasible. I did not yet understand how Marc's alternative definition solves the problem.)

Other points that I remember coming up during our discussions:

- At which granularity should things be repeatable? The execution of an entire program, the execution of individual instructions, ...? (Isaac: The role of an architecture is to help realize a program specification. For example, ISA specifies an order of instructions that need to be preserved to preserve the semantics. In that sense, an architecture is repeatable in that to the programmer it is repeatedly execution instructions in the same order. I think at every granularity things need to be repeatable, it's just what properties at what granularity need to be repeatable?)
- What kind of properties are we interested in? (Marc: At least timing related aspects...) (Isaac: To point out the obvious, execution time is a property that we care about. Thus, if the specification (ISA) of programs

Background: The Tagged Signal Model



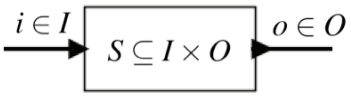
Example of a property this model might have:

Determinism:

$$(i, o_1) \in S \wedge (i, o_2) \in S \Rightarrow o_1 = o_2$$

Determinism is a Boolean property of the model: It either has it or it doesn't.

Example: Turing Machine



$\mathbb{B} = \{\text{true}, \text{false}\}$

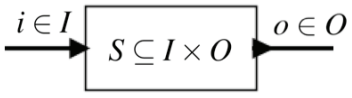
$I = \mathbb{B}^*$

$O = \mathbb{B} \cup \{\varepsilon\}$, where ε represents nontermination.

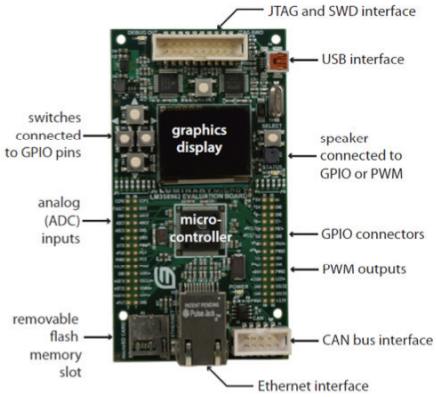
S is a Turing machine.

Then S is determinate.

Model vs. Realization



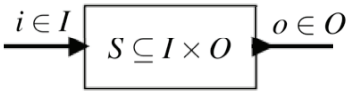
Model



Realization

Is the Realization determinate?

Properties of Models



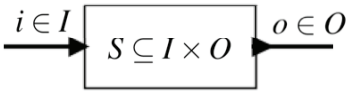
Property: $P \subseteq 2^{I \times O}$

P is a *repeatable property* of S if $\exists\ p \in P$ such that

$$(i,o) \in S \quad \Rightarrow \quad (i,o) \in p \quad \vee \quad \nexists\ o' \in O \text{ where } (i,o') \in p$$

I.e., if an input is mentioned in P, then the corresponding output is allowed. (This definition assumes S is *receptive,* meaning that it has a behavior for all $i \in I$).

Example: Determinacy



Property: $P \subseteq 2^{I \times O}$

Determinacy: P is the set of all functions of form $p: I \rightarrow O$

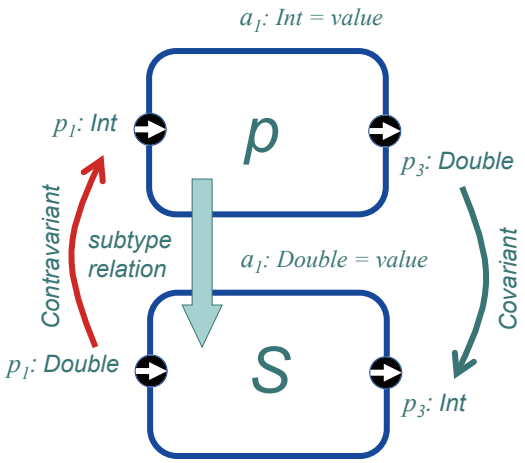
S is a *determinate* if $\exists\ p \in P$ such that

$$(i,o) \in S \quad \Rightarrow \quad (i,o) \in p$$

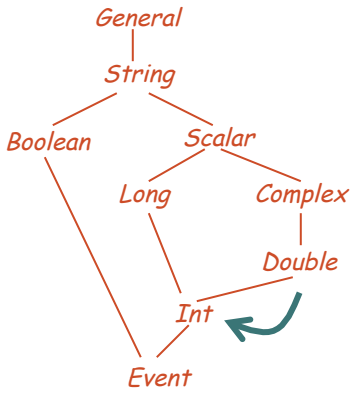
I.e., there is a function in P matching the behavior of S. (Again, for simplicity, we assume S is *receptive*).

Subtype Interpretation (Refinement)

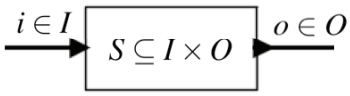
$(i,o) \in S \quad \Rightarrow \quad (i,o) \in p \quad \vee \quad \nexists\, o' \in O$ where $(i,o') \in p$



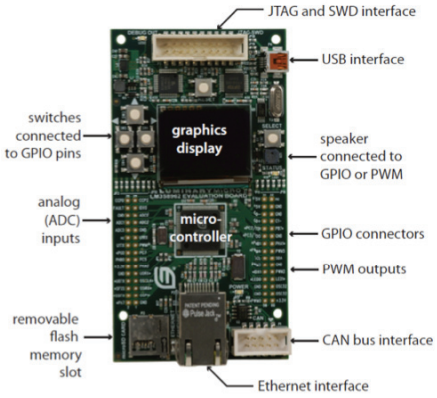
Example of a simple type lattice:



Testing



Model



Realization

Testing is the process of checking that a realization matches a (typically) finite repeatable property of the model:

$$P = \{\{(i_1,o_1),(i_2,o_2),\cdots(i_n,o_n)\}\}$$

Predictability

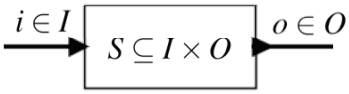
Predictability is the ability to *anticipate* the behavior of a system.

P is a *predictable property* of model S if there is a terminating procedure that proves that P is a repeatable property of S. Specifically:

Given as input a finite description of the model and a finite description of the input or a set of inputs in *I*, a terminating procedure proves that the property is repeatable: $\exists\ p \in P$

$(i,o) \in S \quad \Rightarrow \quad (i,o) \in p \quad \vee \quad \nexists\ o' \in O$ where $(i,o') \in p$

Example: Universal Turing Machine



$\mathbb{B} = \{\text{true}, \text{false}\}$

$I = \mathbb{B}^*$ (input encodes a Turing machine and its input)

$O = \mathbb{B} \cup \{\varepsilon\}$, where ε represents non-termination.

Suppose $I_T \subset I$ is some set of inputs on which the machine terminates. Let

$$P = \{\{(i,o) \mid i \in I_T,\ o \in \mathbb{B}\}\}$$

P is a *repeatable but (often) not predictable* property of S.

Predictability Requires Proof

To *anticipate*, we have a *terminating* procedure that shows that a property holds.

Example: Let A be the set of instructions in an ISA. Then A^* is the set of programs for this ISA.

A program $a \in A$ predictably terminates iff there is a terminating procedure that determines that a terminates.

Real-Time Properties of Programs

Need to augment the model to include timing.

Assume: $I = O = \mathbb{B}^*$

A minimal addition simply augments the output to include the amount of time the computation takes.

$$O' = O \times \mathbb{R}_+$$

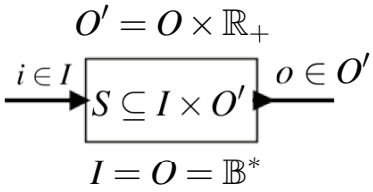
$$i \in I \rightarrow \boxed{S \subseteq I \times O'} \rightarrow o \in O'$$

WCET as a Property

E.g, for all inputs $i \in I$, the computation takes no more than WCET seconds:

$$P = \{\{(i,(o,t)) \mid i \in I \ \wedge \ o \in O \ \wedge \ t \leq \text{WCET}\}\}$$

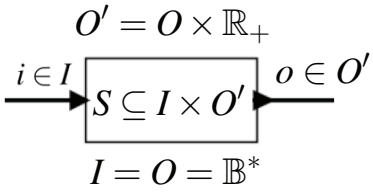
Is this property predictable? Repeatable?



Real-Time Properties of Programs

The timing of programs in all modern programming languages is neither predictable nor repeatable.

The finite description of the system is a program, which is a model rooted in a Turing-Church view of computation. Timing is not in the domain of discourse.

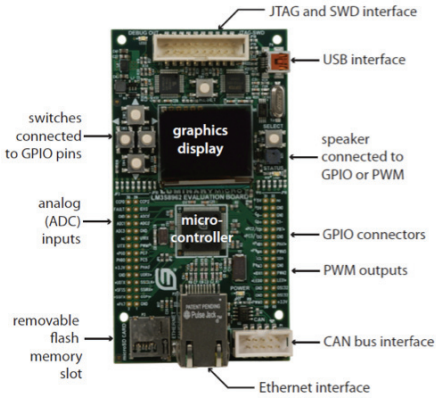


The Standard Practice Today: WCET Analysis

Today, we *augment the model* with *minute* details of the realization

(not just ISA, but how the ISA is realized, what memory technology is used, how much memory of each kind, what I/O hardware is used, exact timing of inputs, etc.)

We can do better!



Realization

Ideas about How to Do Better

I will now talk about ways to augment the notion of an *instruction set architecture* (ISA) to include *some* timing properties in the model.

PRET Machines
Make Timing a Semantic Property of Computers

Precision-Timed (PRET) Machines

Just as we expect reliable logic operations, we should expect repeatable timing.

Timing precision with performance: Challenges:

- Memory hierarchy (scratchpads?)
- Deep pipelines (interleaving?)
- ISAs with timing (deadline instructions?)
- Predictable memory management (Metronome?)
- Languages with timing (discrete events? Giotto?)
- Predictable concurrency (synchronous languages?)
- Composable timed components (actor-oriented?)
- Precision networks (TTA? Time synchronization?)

See S. Edwards and E. A. Lee, "**The Case for the Precision Timed (PRET) Machine**," in the *Wild and Crazy Ideas* Track of the *Design Automation Conference* (DAC), June 2007.

Now: Focus on Extending the ISA with Temporal Semantics

Some possible capabilities in an ISA:

- [V1] Execute a block of code taking at least a specified *time* [Ip & Edwards, 2006]
- [V2] Do [V1], and then conditionally branch if the specified *time* was exceeded.
- [V3] Do [V1], but if the specified *time* is exceeded during execution of the block, branch immediately to an exception handler.
- [V4] Execute a block of code taking exactly the specified *time*. MTFD

Here, *time* may be literal (seconds) or abstract (cycles).

Focus on Extending the ISA with Temporal Semantics

How to use these capabilities:

- [V1] is useful for best-effort control over timing.
- [V2] is useful for best-effort control over timing with adaptation when specified timing cannot be met.
- [V3] is useful when timing constraints are expected to be met, but we cannot prove that they will be met (timing is repeatable but not predictable).
- [V4] is useful only if timing is predictable.

PRET Assembly Instructions Supporting these Four Capabilities

set_time %r, <val>

– loads current time + <val> into %r

delay_until %r

– stall until current time >= %r

branch_expired %r, <target>

– branch to target if current time > %r

exception_on_expire %r, <id>

– arm processor to throw exception <id> when current time > %r

deactivate_exception <id>

– disarm the processor for exception <id>

MTFD %r

– stall and continue when current time = %r

Controlled Timing in Assembly Code

[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
```


Other variations

[V2]-[V4] could all have a variant that does not control the *minimum* execution time of the block of code, but only controls the *maximum*.

[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
```

Application: Timed Loops

Fixed Period

```
set_time r1, 1s
loop:
// Code block
delay_until r1
r1 = r1 + 1s
b loop
```

Lower bound for each iteration

```
set_time r1, 1s
loop:
// Code block
delay_until r1
set_time r1, 1s
b loop
```

The two loops above have different semantics:

- On the left, a timing overrun is expected to be made up on subsequent iterations.
- On the right, each iteration takes 1 second (or more, if it overruns).

Timed Loop with Exception Handling

Exact execution time (no jitter)

```
set_time r1, 1s
exception_on_expire r1, 0
loop:
// Code block
deactivate_exception 0
delay_until r1
r1 = r1 + 1s
exception_on_expire r1, 0
b loop
```

This code takes exactly 1 second to execute each iteration. If an iteration takes more than 1 second, then as soon as its time expires, the iteration is aborted and an exception handler is activated.

Exact Timed Loop

Exact execution time (no jitter)

```
set_time r1, 1s
loop:
// Code block
MTFD r1
r1 = r1 + 1s
b loop
```

This code takes exactly 1 second to execute each iteration.

How to implement this?

1. At run time, MTFD = delay_until
2. At compile time (or load time), the code does not compile (or load) if meeting the deadline cannot be proved by static analysis.

Exporting the Timed Semantics to a Low-Level Language (like C)

```
tryin (500ms) {
   // Code block
} expired {
    patchup();
}
```

```
set_time r1, 500ms
// Code block
branch_expired r1, patchup
```

This realizes variant 2, "late miss dectection."

The code block will execute to completion.
If 500ms has passed, then the patchup procedure will run.

Exporting the Timed Semantics to a Low-Level Language (like C)

```
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)
}
```

This realizes variant 3, "immediate miss detection," using setjmp and longjmp to handle timing exceptions. setjmp() returns 0 when directly invoked, and returns non zero when invoked by longjmp().

If the code block takes longer than 500ms to run, then exception 0 will be thrown, and the handler code will run longjmp, which will return control flow to setjmp, but returning non zero. The else statement will then be run, causing the panic procedure to be called.

This pseudocode is neither C-level nor assembly, but is meant to explain an assembly-level implementation.

Lee et. al, Berkeley 31

Variant with Exact Execution Times: tryfor

```
tryfor (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
  delay_until r1
} else {
    panic();
}

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

This is the same, except for the added delay_until

Lee et. al, Berkeley 32

Summary of ISA extensions

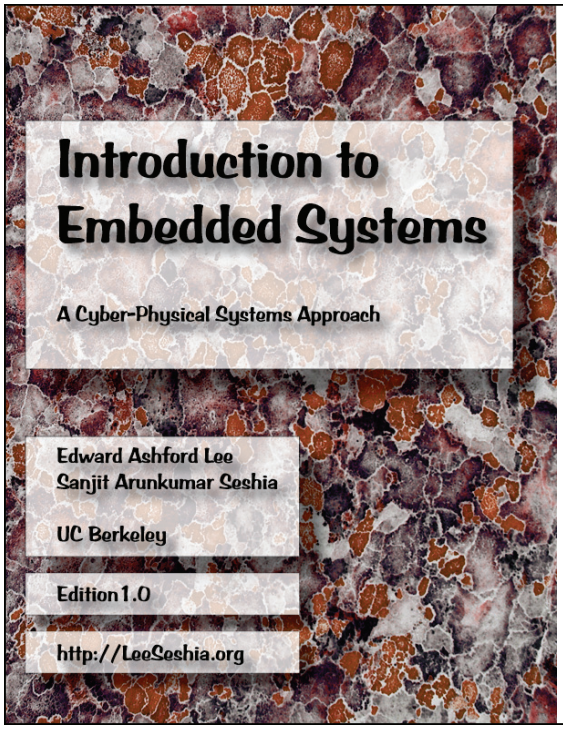
- [V1] Execute a block of code taking at least a specified *time* [Ip & Edwards, 2006]
- [V2] Do [V1], and then conditionally branch if the specified *time* was exceeded.
- [V3] Do [V1], but if the specified *time* is exceeded during execution of the block, branch immediately to an exception handler.
- [V4] Execute a block of code taking exactly the specified *time*. MTFD

Variants:
- For V2 & V3, may not impose minimum execution time.
- *Time* may be literal (seconds) or abstract (cycles).

Conclusions

Today, timing is a property only of realizations of systems.
Tomorrow, timing will be a semantic property of models.

Raffaello Sanzio da Urbino – The Athens School



New Text: Lee & Seshia: Introduction to Embedded Systems - A Cyber-Physical Systems Approach

http://LeeSeshia.org/

This book strives to identify and introduce the durable intellectual ideas of embedded systems as a technology and as a subject of study. The emphasis is on modeling, design, and analysis of cyber-physical systems, which integrate computing, networking, and physical processes.

Lee et. al, Berkeley 35