Lecture
Specifications via Timed ASMs
Requirements
Text: Requirements Engineering Management Handbook

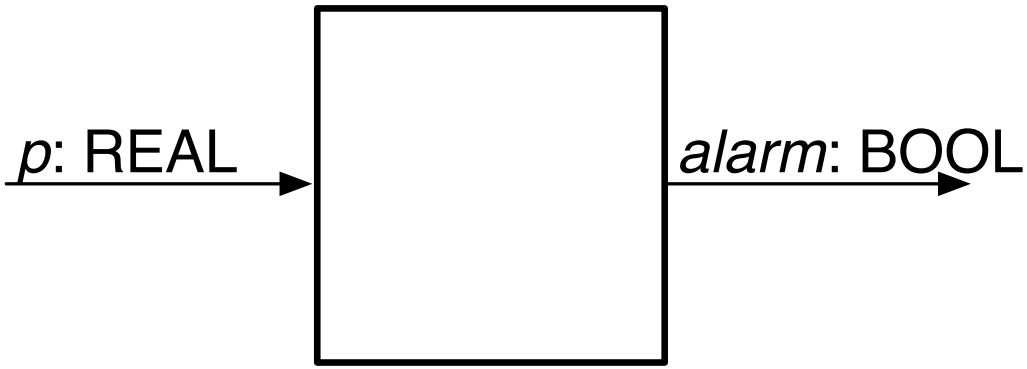
EECS4312 Software Requirements Engineering
Fall 2015

Jonathan Ostroff
York University

LASSONDE SCHOOL OF ENGINEERING

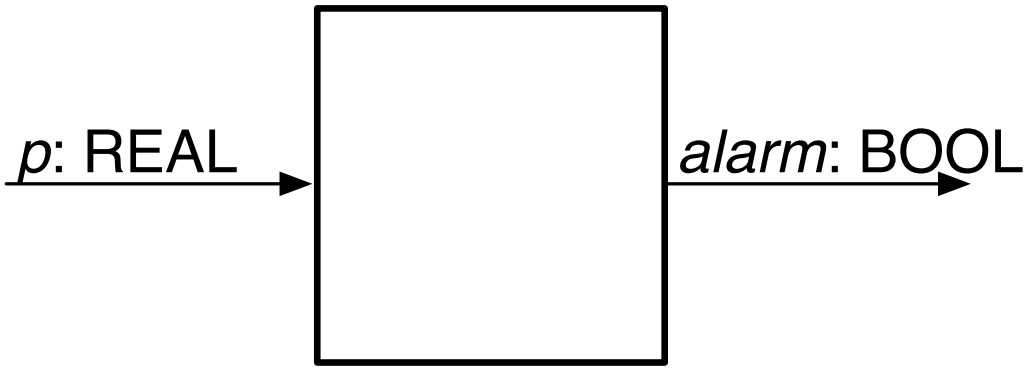
Abstract State Machines

Consider a device with a monitored variable pressure $p : \mathbb{R}$ and controlled variable *alarm* as the output as shown in the figure.



Abstract State Machines

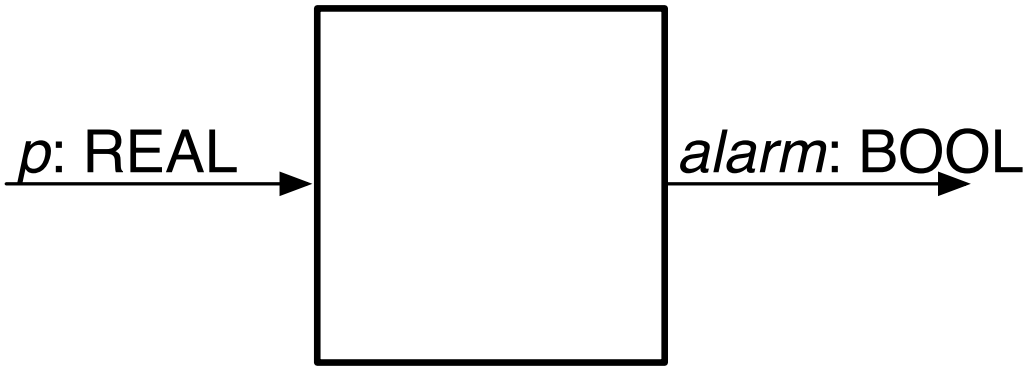
Consider a device with a monitored variable pressure $p : \mathbb{R}$ and controlled variable *alarm* as the output as shown in the figure.



Function Table

Abstract State Machines

Consider a device with a monitored variable pressure $p : \mathbb{R}$ and controlled variable *alarm* as the output as shown in the figure.



Function Table

	alarm
$p \leq normal$	*False*
$normal < p < hi$	No Change
$p \geq hi$	*True*

Abstract State Machines

	alarm
$p \leq normal$	*False*
$normal < p < hi$	No Change
$p \geq hi$	*True*

- The function table specifies the output *alarm* in terms of the input p (and real-valued constants *normal* and *hi*).

Abstract State Machines

	alarm
$p \leq normal$	*False*
$normal < p < hi$	No Change
$p \geq hi$	*True*

- The function table specifies the output *alarm* in terms of the input p (and real-valued constants *normal* and *hi*).
- The function table specifies when the output *alarm* will go high, i.e. generate an alarm signal. "No Change" means that *alarm* stays the same as in the previous state.

Abstract State Machines

	alarm
$p \leq normal$	*False*
$normal < p < hi$	No Change
$p \geq hi$	*True*

- The function table specifies the output *alarm* in terms of the input *p* (and real-valued constants *normal* and *hi*).
- The function table specifies when the output *alarm* will go high, i.e. generate an alarm signal. "No Change" means that *alarm* stays the same as in the previous state.
- In validating such specifications we have to deal with an infinite number of possible inputs (some slice of the real line representing pressures), as opposed to digital circuits such as the majority voting circuit or the Date Validity function table.

Abstract State Machines

	alarm
$p \leq normal$	*False*
$normal < p < hi$	No Change
$p \geq hi$	*True*

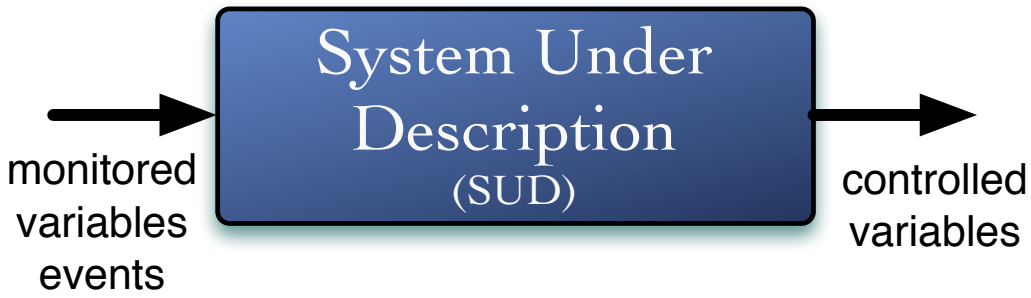
- The function table specifies the output *alarm* in terms of the input *p* (and real-valued constants *normal* and *hi*).
- The function table specifies when the output *alarm* will go high, i.e. generate an alarm signal. "No Change" means that *alarm* stays the same as in the previous state.
- In validating such specifications we have to deal with an infinite number of possible inputs (some slice of the real line representing pressures), as opposed to digital circuits such as the majority voting circuit or the Date Validity function table.
- Furthermore, how would we specify "No Change" from the previous time an input was received?

Abstract State Machines

	alarm
$p \leq normal$	*False*
$normal < p < hi$	No Change
$p \geq hi$	*True*

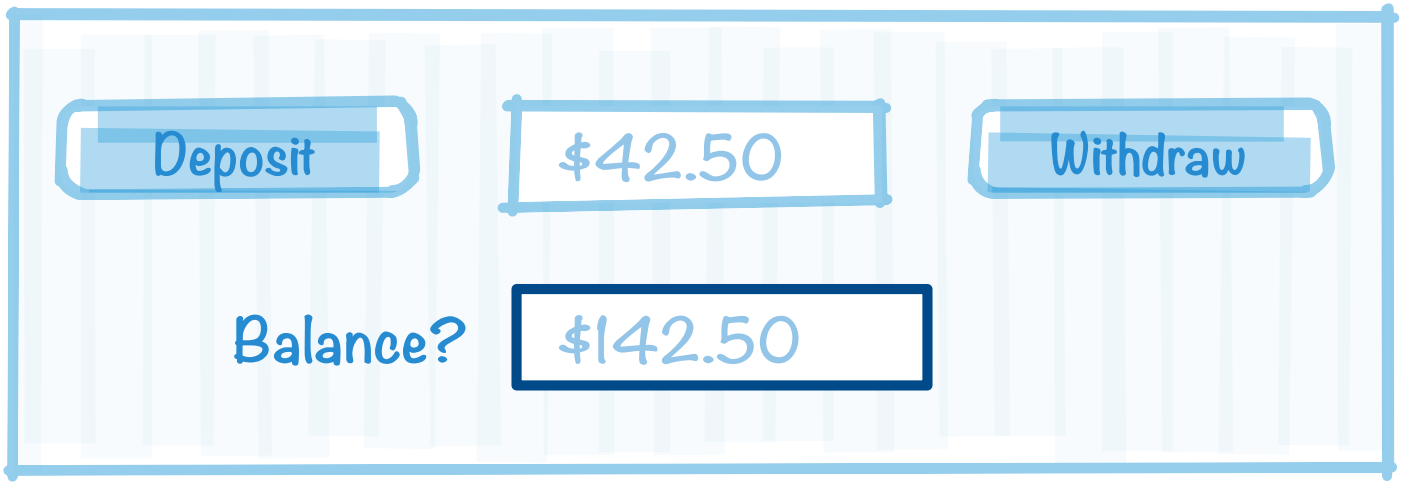
- The function table specifies the output *alarm* in terms of the input *p* (and real-valued constants *normal* and *hi*).
- The function table specifies when the output *alarm* will go high, i.e. generate an alarm signal. "No Change" means that *alarm* stays the same as in the previous state.
- In validating such specifications we have to deal with an infinite number of possible inputs (some slice of the real line representing pressures), as opposed to digital circuits such as the majority voting circuit or the Date Validity function table.
- Furthermore, how would we specify "No Change" from the previous time an input was received?
- No Change (NC) means we need a state machine that has a present state and a previous state.

Monitored and Controlled Variables



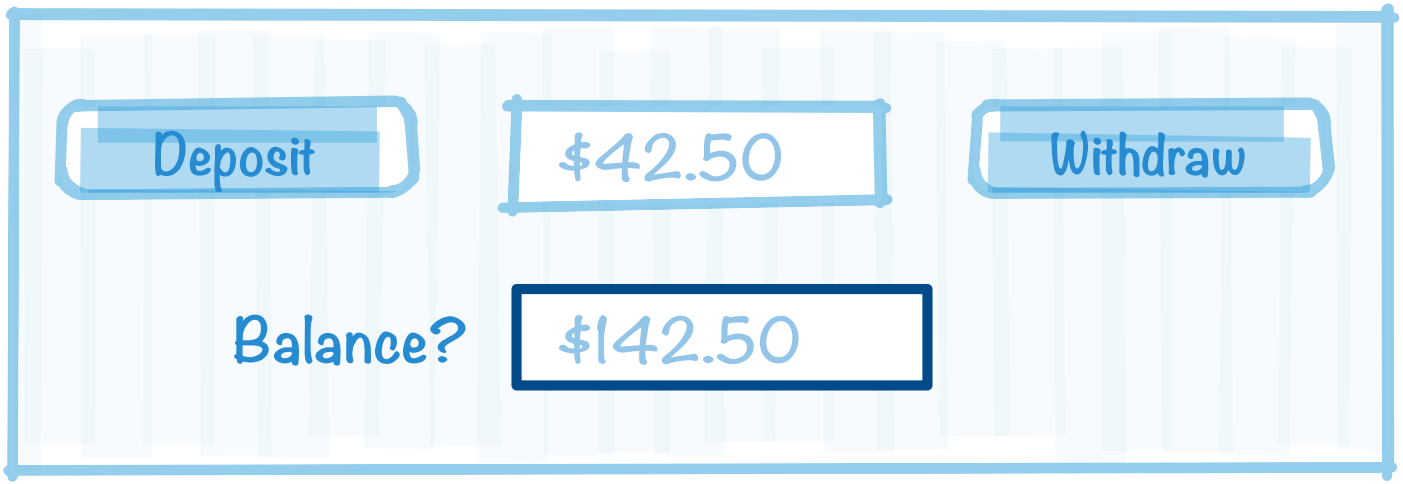
- What are the monitored events and variables?
- What are the controlled variables?
- What is the relationship between monitored variables and controlled variables

Bank ATM Requirements



LASSONDE SCHOOL OF ENGINEERING

Bank ATM Requirements



R1 A user shall be able to deposit and withdraw money at the ATM

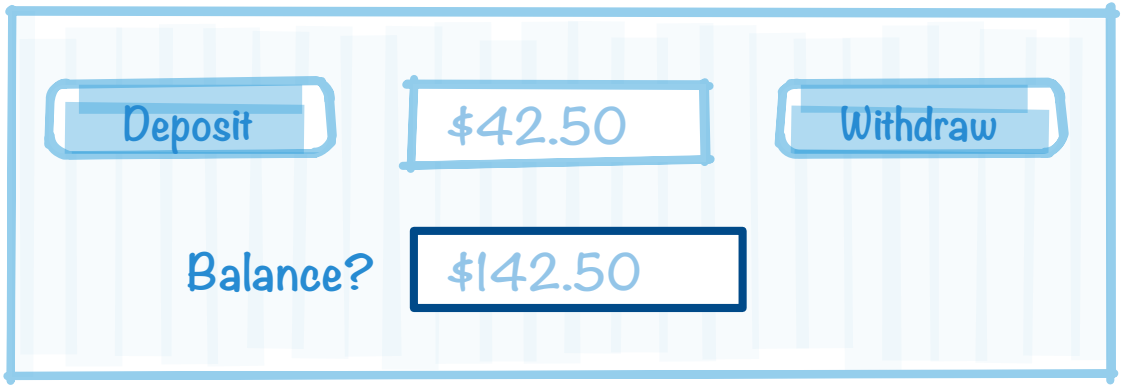
R2 The current balance, total deposits and total withdrawals are displayed at the ATM

R3 The balance shall never be negative

Grammar: What are the input operations?

R1

A user shall be able to deposit and withdraw money at the ATM

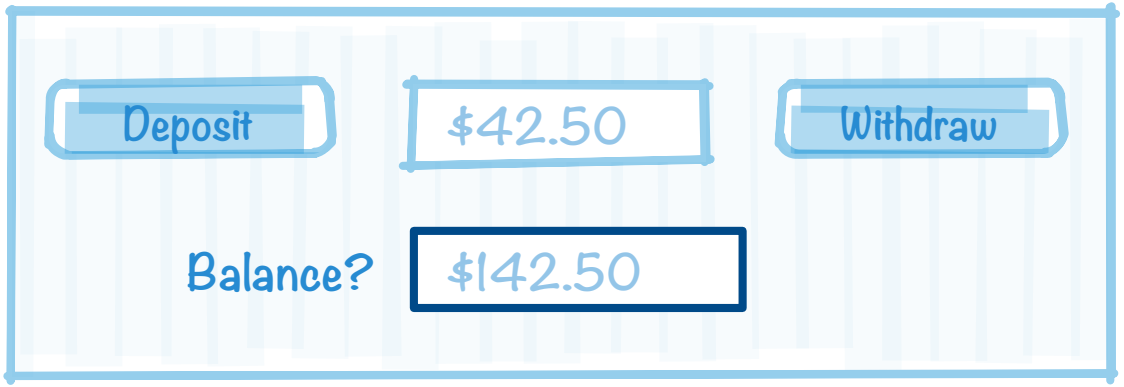


LASSONDE SCHOOL OF ENGINEERING

Grammar: What are the input operations?

R1

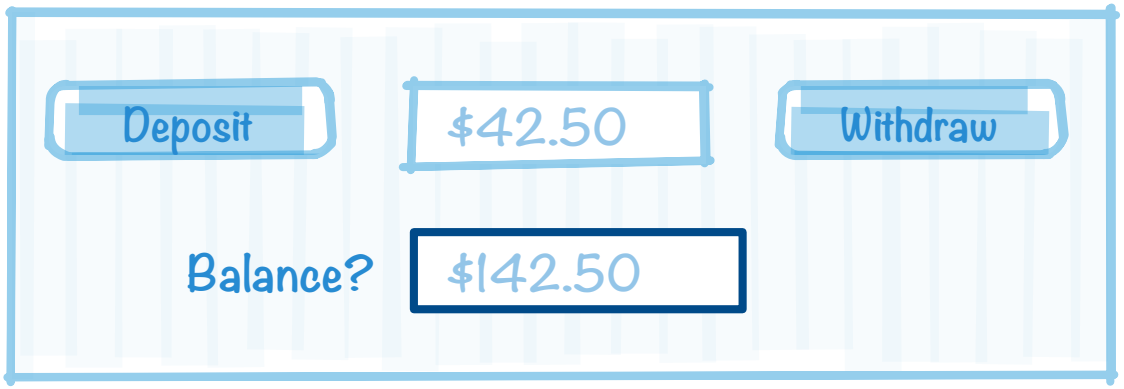
A user shall be able to deposit and withdraw money at the ATM



bank-definitions.txt

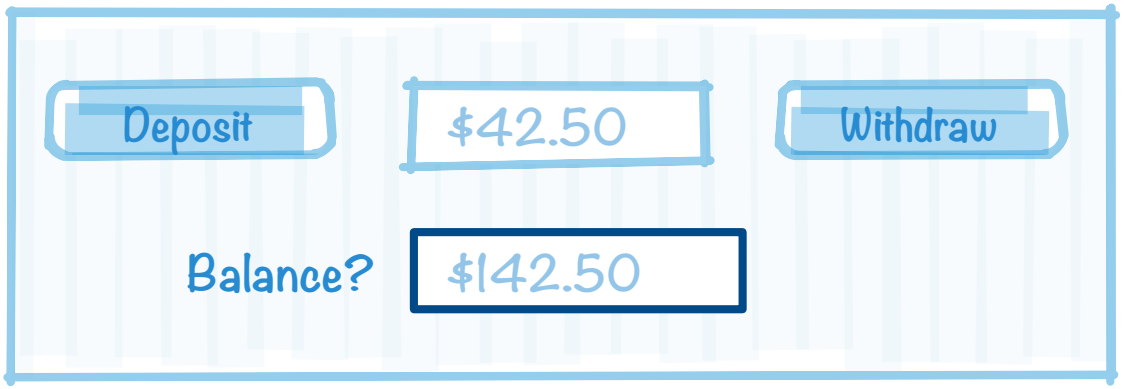
```
system small_bank
nothing
deposit(v: REAL)
withdraw(v: REAL)
```

Abstract user interface




```
system small_bank
nothing
deposit(v: REAL)
withdraw(v: REAL)
```

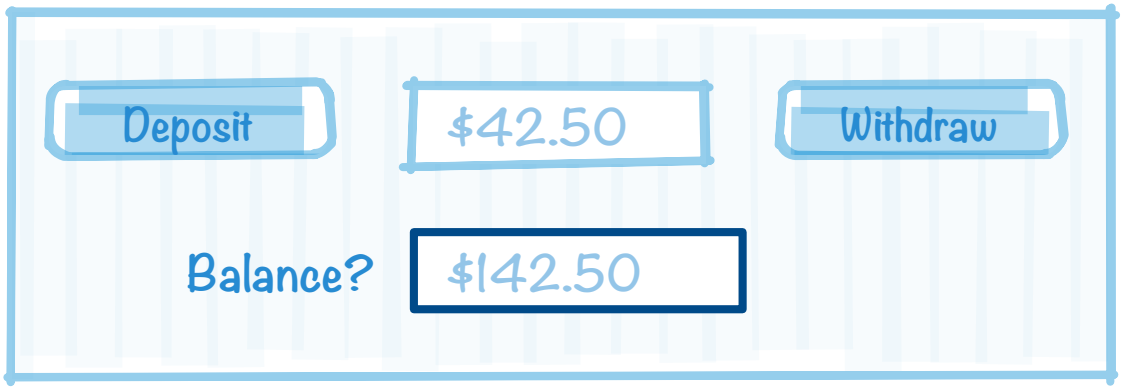
Abstract user interface




```
system small_bank
nothing
deposit(v: REAL)
withdraw(v: REAL) -- v can be zero or negative
```


LASSONDE SCHOOL OF ENGINEERING

Abstract user interface




```
system small_bank
nothing
deposit(v: REAL)
withdraw(v: REAL) -- v can be zero or negative
```

Abstract UI

The grammar defined by `bank-definitions.txt` describes an abstract user interface without the need to prematurely decide on an actual concrete UI.

Output: Controlled variables

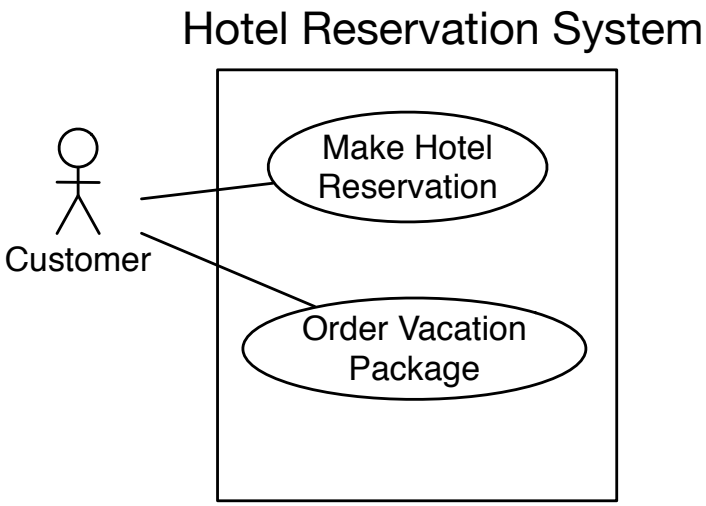
R2

The current balance, total deposits and total withdrawals are displayed at the ATM

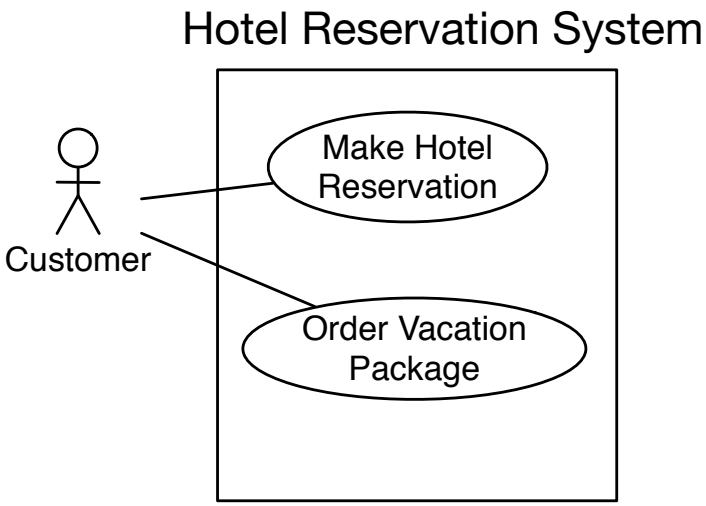
Abstract state (from which the Output is derived)

Variable	Type	Description/Units
b	$\mathbb{R}$	balance in dollars
d	$\mathbb{R}$	total deposits in dollars
w	$\mathbb{R}$	total withdrawals

UML Use Cases



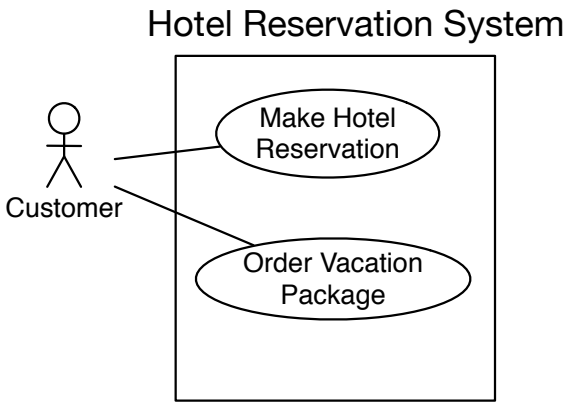
UML Use Cases



Textual Use Case

1. Enter (city, arrival, departure, room-type) and click Search
2. System: Display list of hotels
3. Click on hotel's logo to read details
4. System: Display hot details
5. Click Book Now
6. System: Display payment form
7. Enter customer details, billing information and click Submit
8. System: Validate billing information and display result

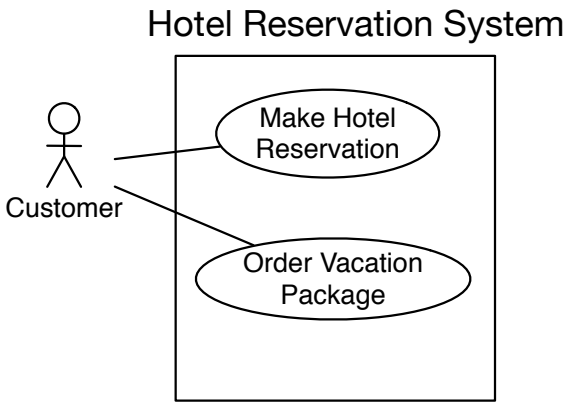
UML Use Cases



What is a Use Case?

- A use case is a list of steps/events that describe the interactions between an external Actor and a System (software or hardware), to achieve a customer goal

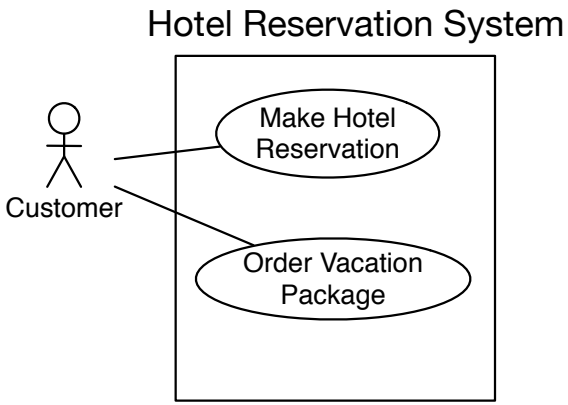
UML Use Cases



What is a Use Case?

- A use case is a list of steps/events that describe the interactions between an external Actor and a System (software or hardware), to achieve a customer goal
- An Actor can be a human, time or an an external entity (such as a sensor, actuator, database, etc.)

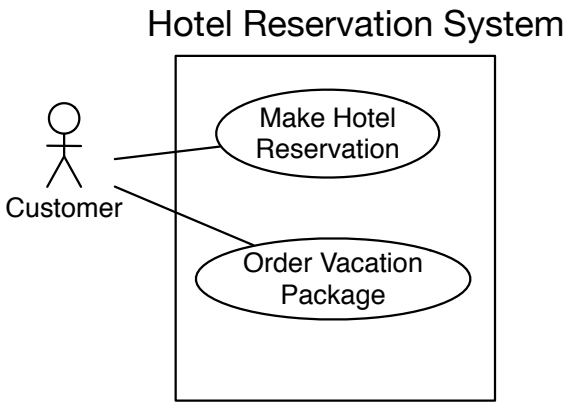
UML Use Cases



What is a Use Case?

- A use case is a list of steps/events that describe the interactions between an external Actor and a System (software or hardware), to achieve a customer goal
- An Actor can be a human, time or an an external entity (such as a sensor, actuator, database, etc.)
- Use case analysis is an important and valuable requirements analysis technique that has been widely used in modern software engineering

UML Use Cases



What is a Use Case?

- A use case is a list of steps/events that describe the interactions between an external Actor and a System (software or hardware), to achieve a customer goal
- An Actor can be a human, time or an an external entity (such as a sensor, actuator, database, etc.)
- Use case analysis is an important and valuable requirements analysis technique that has been widely used in modern software engineering
- It is part of the UML standard that you must master.
- See `https://wiki.eecs.yorku.ca/project/eiffel/_media/bon:uml.pdf`

Use Case 1: ATM Deposit and withdraw

```
  init
-> deposit(100)
  b: 100
  d: 100
  w: 0
-> withdraw(50)
  b: 50
  d: ??
  w: ??
-> withdraw(50)
  b: 0
  d: ??
  w: 100
 -> withdraw(1)
   b: ??
   d: ??
   w: ??
```

Use Case 1: Deposit and withdraw

```
  init
-> deposit(100)
  b: 100
  d: 100
  w: 0
-> withdraw(50)
  b: 50
  d: 100
  w: 50
-> withdraw(50)
  b: 0
  d: 100
  w: 100
 -> withdraw(1)
   b: ??
   d: 100
   w: 100
```

Use Case 2: Nothing, deposit and withdraw

```
  init
-> deposit(100)
  b: 100
  d: 100
  w: 0
-> withdraw(50)
  b: 50
  d: 100
  w: 50
-> nothing
  b: ??
  d: ??
  w: ??
 -> deposit(125)
   b: ??
   d: ??
   w: ??
```

Use Case 2: Nothing, deposit and withdraw

```
  init
-> deposit(100)
  b: 100
  d: 100
  w: 0
-> withdraw(50)
  b: 50
  d: 100
  w: 50
-> nothing
  b: NC -- how to express no change?
  d: 100
  w: 50
 -> deposit(125)
   b: ??
   d: ??
   w: ??
```

Other Operations?

Other operations

How about transferring from one account to another?

Other Operations?

Other operations

How about transferring from one account to another?

```
system bank
new(id: STRING)
deposit(id: STRING; amount: REAL)
withdraw(id: STRING; amount: REAL)
transfer(to: STRING; from: STRING; amount: REAL)
```

Other Use Cases?

Other Use Cases

How many use cases can we come up with?

Complete Specifications?

What we need is a specification that is

- complete

Other Use Cases?

Other Use Cases

How many use cases can we come up with?

Complete Specifications?

What we need is a specification that is

- complete
- Can be used to validate all use cases

Specification of ATM via a Function Table

			b	d	w	report
	nothing		**old** b	**old** d	**old** w	**old** report
	deposit(v)					
	withdraw(v)					

Function Table: What does "old' mean?'

		b	d	w	report
	nothing	**old** b	**old** d	**old** w	**old** report
	deposit(v)				
	withdraw(v)				

What does old mean?

In an Eiffel routine it is clear.

LASSONDE SCHOOL OF ENGINEERING

Function Table: What does "old' mean?'

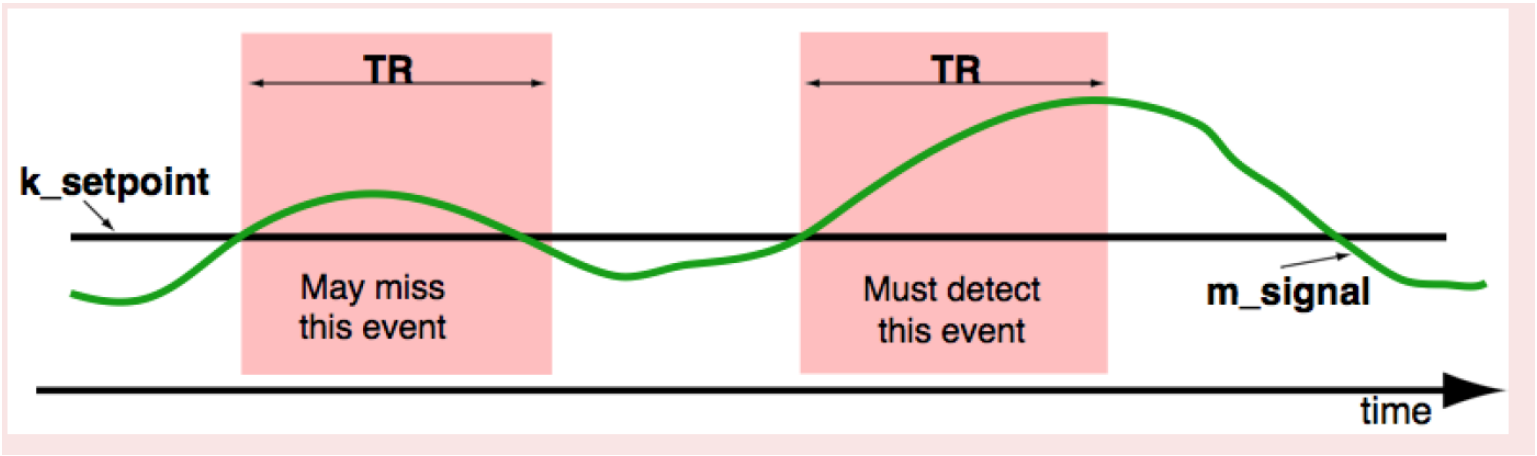
			b	d	w	report
	nothing		**old** b	**old** d	**old** w	**old** report
	deposit(v)					
	withdraw(v)					

What does old mean?

In an Eiffel routine it is clear.

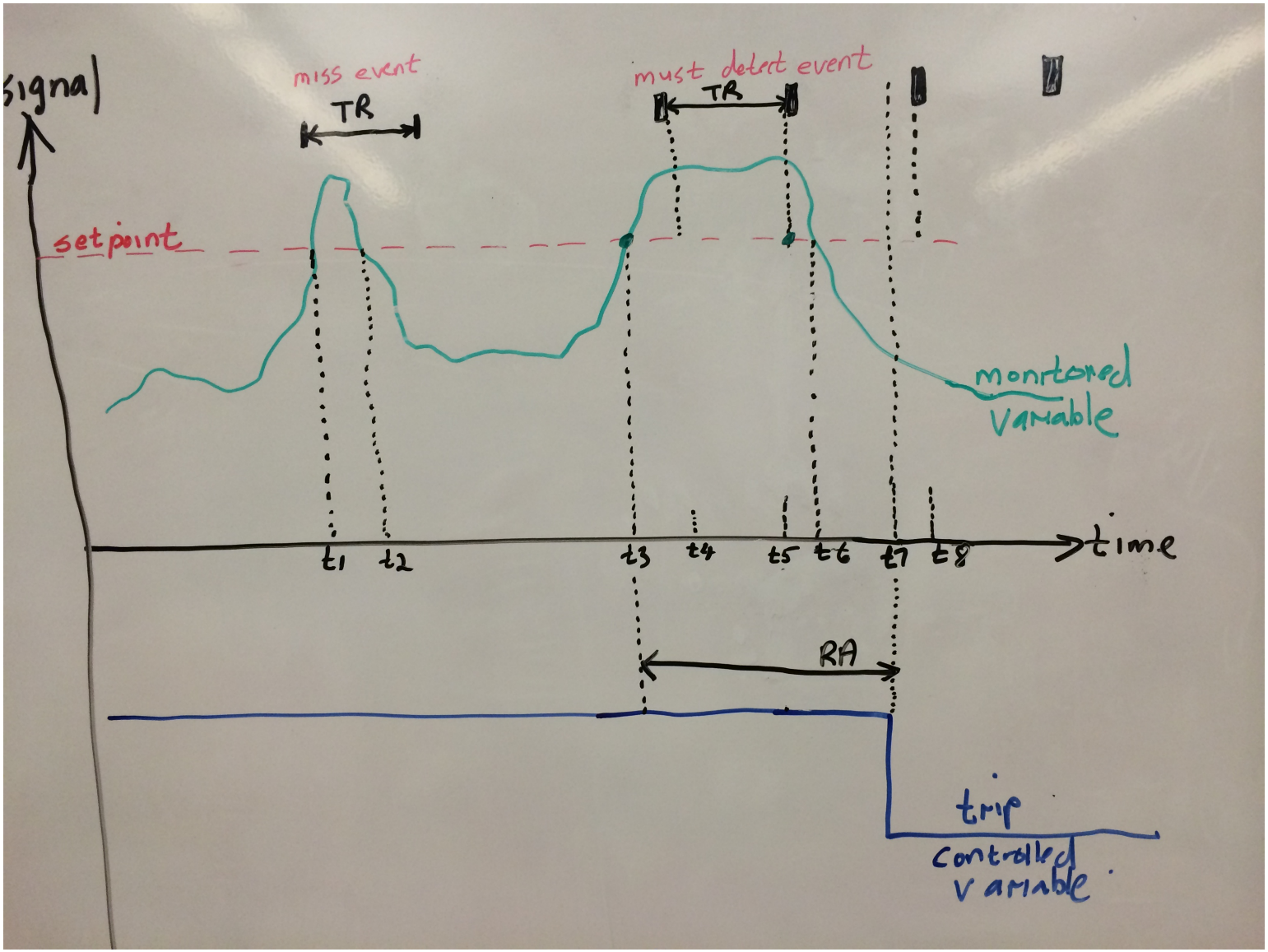
But we are dealing with console input/output modelling an abstract user interface!

Timing Resolution TR



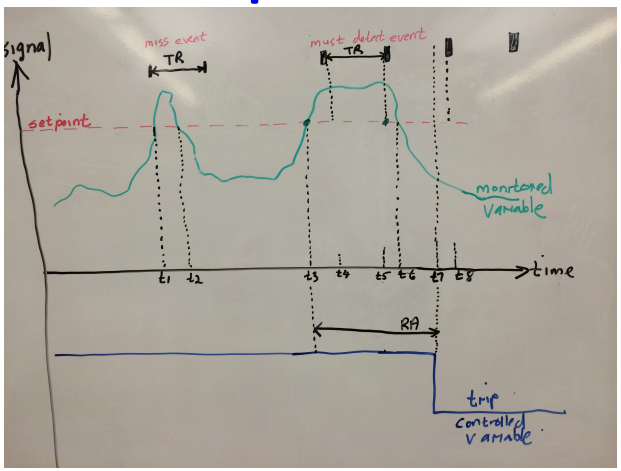
LASSONDE SCHOOL OF ENGINEERING

Timing Resolution (TR) and Response Allowance (RA)



LASSONDE SCHOOL OF ENGINEERING

Relationship between TR and RA

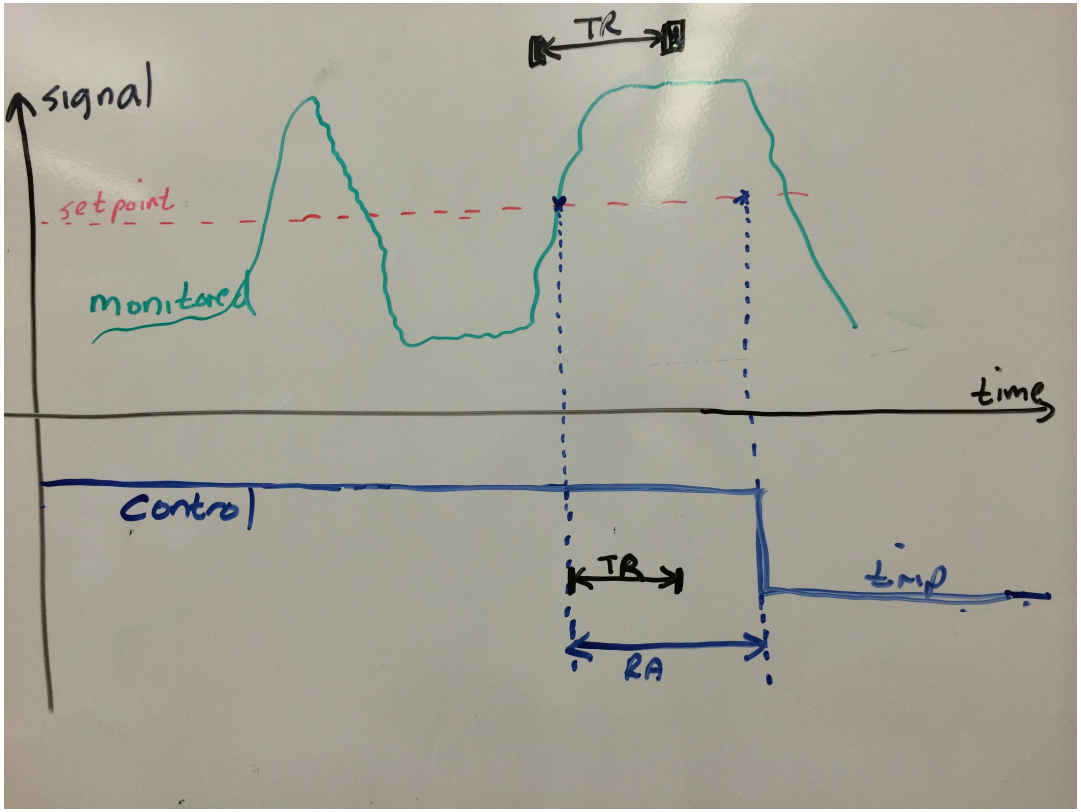


Relationship

- The timing resolution (TR) for a time continuous monitored variable is the minimum time duration of an initiating event dependent on that monitored variable for which the application must guarantee that it will detect that event. Thus, the TR is also an indication of the maximum time interval that the control computer can allow between successive sampling instances for that stimulus.
- The RA is measured from the time the measured event actually occurred in the physical domain, until the time the value of the controlled variable is generated and crosses the application boundary into the physical domain.

LASSONDE SCHOOL OF ENGINEERING

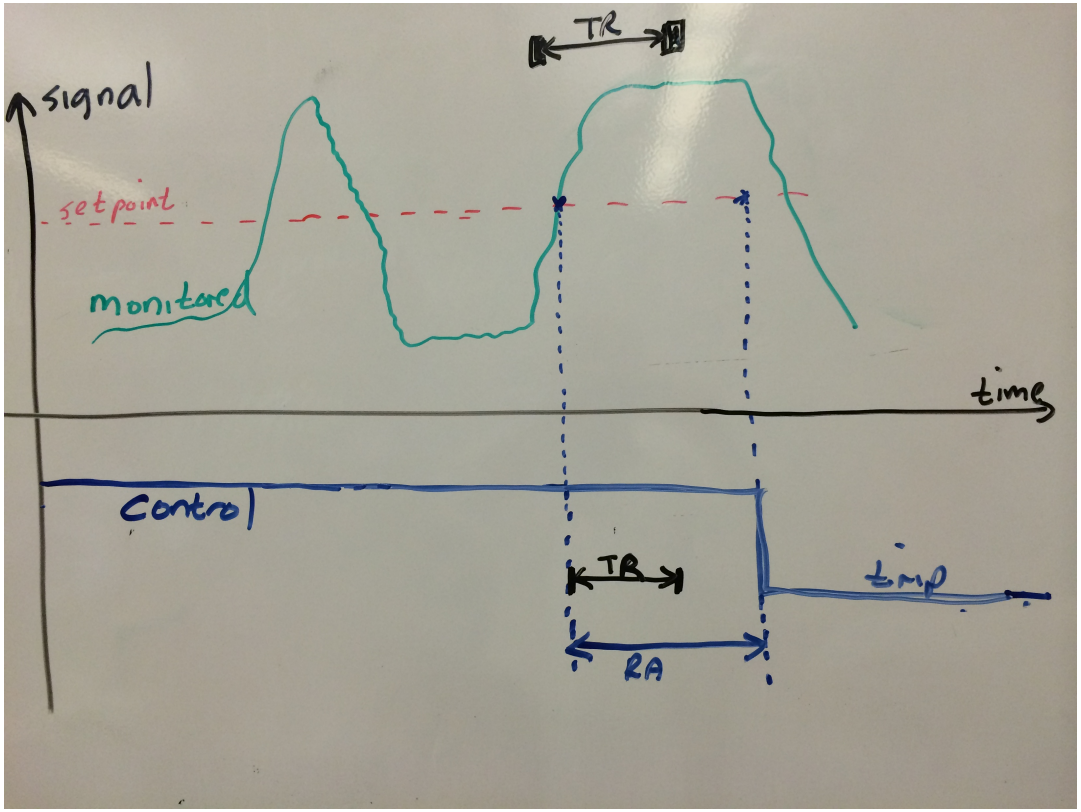
Relationship between TR and RA



What is the relationship between TR and RA?

- Timing Resolution (TR) and Response Allowance (RA)

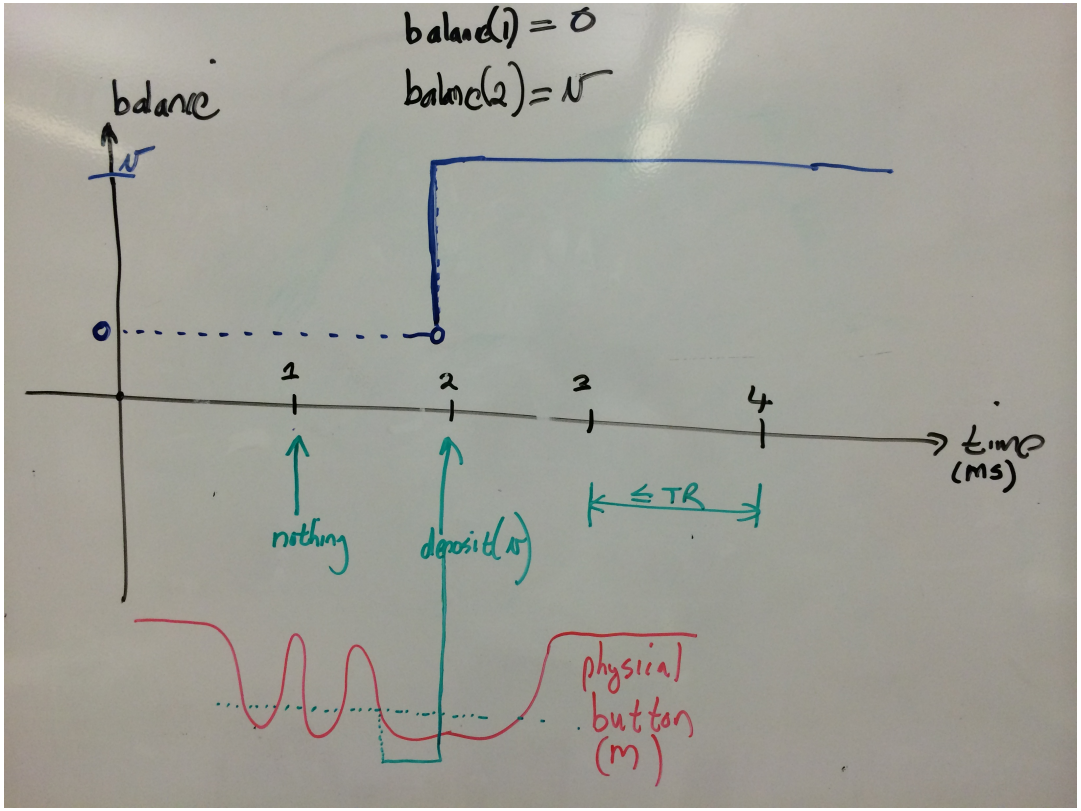
Relationship between TR and RA



What is the relationship between TR and RA?

- Timing Resolution (TR) and Response Allowance (RA)
- $TR < RA$

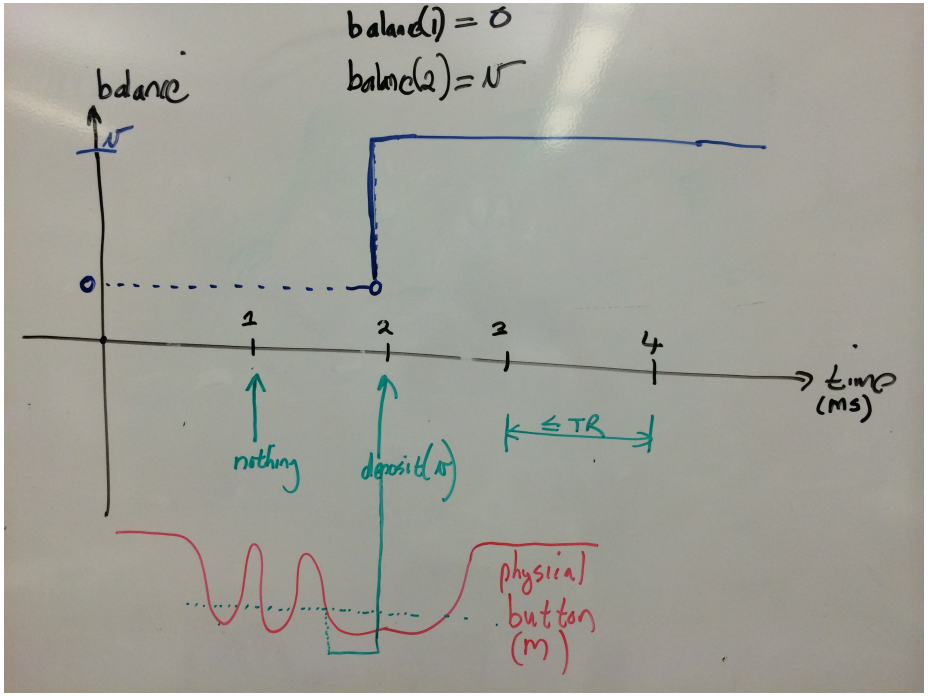
Discrete time i (relative to real-time)



Time i

- $i = 1$ is at the same time as real-time $t = .001s$
- $i = 2$ is at the same time as real-time $t = .002s$
- $balance(0) = 0$, $balance(1) = 0$, $balance(2) = v$
- $i = 2 \Rightarrow balance(i) = balance(i-1) + v$

Physical button (red)



Separation of concerns

- At a hardware level, some circuits might take care of dealing with the electrical signal.
- Alternatively, the graphic runtime (e.g. Swing, WinForms, Eiffel Vision, Android View framework) and the OS might take care of button presses, swipes and other input actions.
- You might also have to take care of the (red) physical signal in software. If so, you can specify a hardware hiding module.

Meaning of "old"– in Timed ASM Theory

Meaning of "old"– in Timed ASM Theory

```
DTIME : TYPE = nat
% psuedo, digitized real time
RTIME : TYPE =
 {t : nnreal | (EXISTS (i : DTIME) : t = i * delta)}
% conversions
r2d(t: RTIME): DTIME = t / delta
d2r(i: DTIME): RTIME = i * delta
```

Meaning of "old"– in Timed ASM Theory

```
DTIME : TYPE = nat
% psuedo, digitized real time
RTIME : TYPE =
 {t : nnreal | (EXISTS (i : DTIME) : t = i * delta)}
% conversions
r2d(t: RTIME): DTIME = t / delta
d2r(i: DTIME): RTIME = i * delta
```

- Convert digital instance i to real-time:
  $d2r(i) = i * delta = i * .001s = .001 * i$ seconds,
  where *delta* is the sampling rate which must be less than the TR.
- i is the instance at which the press of the deposit button signal (generated by the user in the environment) crosses the application boundary and is detected by the computer.
- $deposit(i)$ is thus the instant at which the deposit event is detected by the computer at the monitored input.

Meaning of "old"

- i is the instance at which the press of the deposit button signal (generated by the user in the environment) crosses the application boundary and is detected by the computer.
- $i-1$ is the sampling instant before i
- In a requirements documents, the "idealized" computer can generate an instantaneous change in the controlled variables at instant i based on the monitored input at i, $i-1$ and previous sampling instants.
- However, when the computer system is implemented, we may allow RA seconds to elapse from the time the button was clicked in the environment.

Meaning of "old"

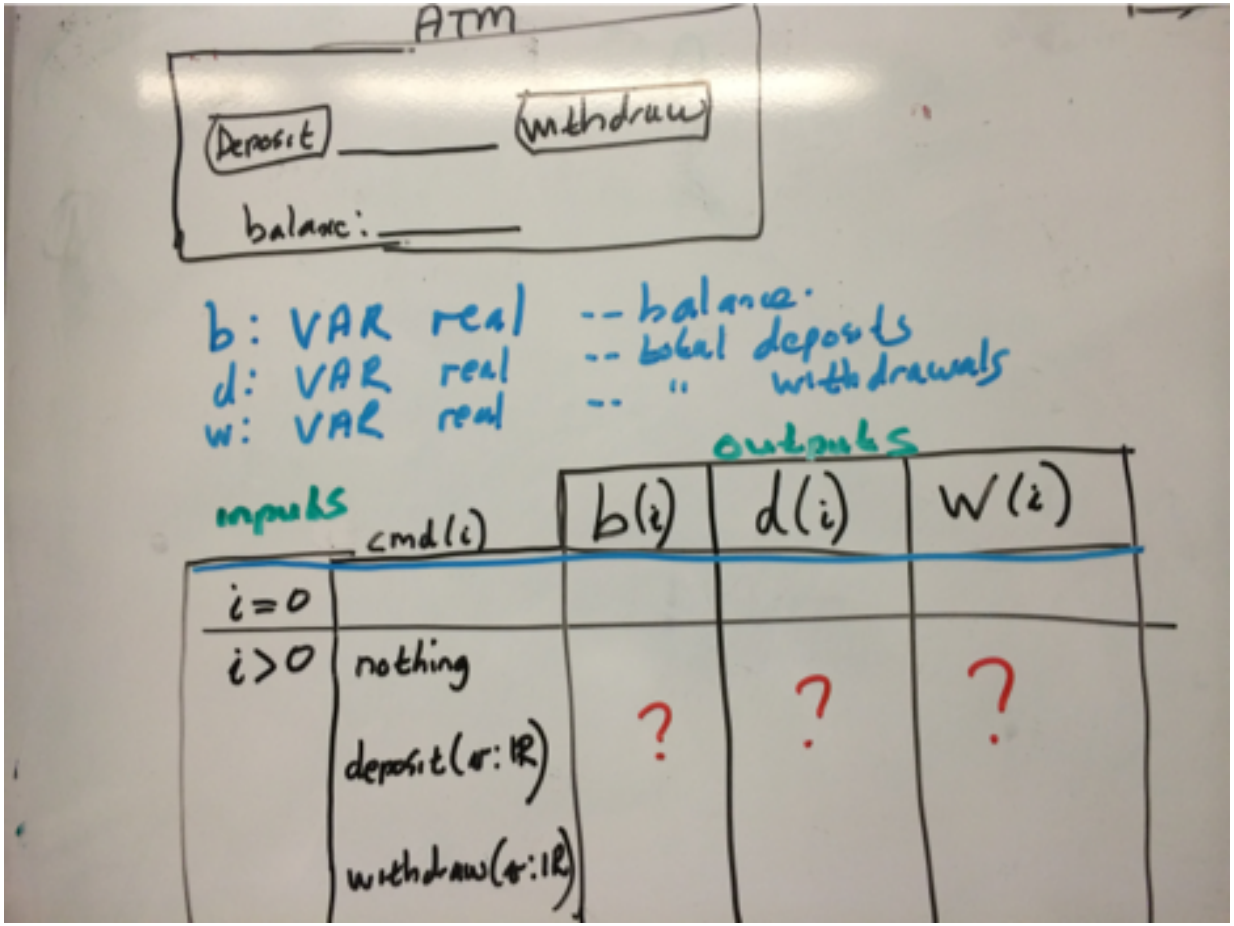
- i is the instance at which the press of the deposit button signal (generated by the user in the environment) crosses the application boundary and is detected by the computer.
- $i-1$ is the sampling instant before i
- In a requirements documents, the "idealized" computer can generate an instantaneous change in the controlled variables at instant i based on the monitored input at i, $i-1$ and previous sampling instants.
- However, when the computer system is implemented, we may allow RA seconds to elapse from the time the button was clicked in the environment.

Meaning of old

old *balance* is thus: $balance(i-1)$, i.e. the value of the balance in the sampling instant before i

How do we specify Bank as a Function Table using T-ASM?

How do we specify Bank as a Function Table using T-ASM?



LASSONDE SCHOOL OF ENGINEERING

Revised Function Table

Conditions

- What is c1?
- Why not just c1 and $\neg$ c1.
- What is c2 and c3?
- What is NC?

monitored inputs: $cmd(i)$				$b(i)$	$d(i)$	$w(i)$	$r(i)$
$i = 0$				0	0	0	ok
$i > 0$	nothing			$b(i-1)$	$d(i-1)$	$w(i-1)$	$r(i-1)$
	dep(v)	$v > 0$		$b(i-1) + v$	$d(i-1) + v$	$w(i-1)$	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	c1		$b(i-1) - v$	$d(i-1)$	$w(i-1) + v$	ok
		$\neg$ c1	c2	NC	NC	NC	er1
			c3	NC	NC	NC	er2

Abbreviations help to fit the function table on one page

Revised Function Table

Conditions

$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i-1) - v \geq 0)$

$c2 \mathrel{\widehat{=}} v \leq 0$

$c3 \mathrel{\widehat{=}} b(i-1) - v < 0$

NC means No Change

monitored inputs: $cmd(i)$				$b(i)$	$d(i)$	$w(i)$	$r(i)$
$i = 0$				0	0	0	ok
$i > 0$	nothing			$b(i-1)$	$d(i-1)$	$w(i-1)$	$r(i-1)$
	dep(v)	$v > 0$		$b(i-1) + v$	$d(i-1) + v$	$w(i-1)$	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	c1		$b(i-1) - v$	$d(i-1)$	$w(i-1) + v$	ok
		$\neg$ c1	c2	NC	NC	NC	er1
			c3	NC	NC	NC	er2

What we have specified is a finite state machine enriched with abstract state

Function Table

Completeness

Is the function table complete and disjoint?

$$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i{-}1) - v \geq 0)$$
$$c2 \mathrel{\widehat{=}} v \leq 0$$
$$c3 \mathrel{\widehat{=}} b(i{-}1) - v < 0$$

LASSONDE SCHOOL OF ENGINEERING

Function Table

LASSONDE SCHOOL OF ENGINEERING

Completeness

Is the function table complete and disjoint?

$$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i-1) - v \geq 0)$$
$$c2 \mathrel{\widehat{=}} v \leq 0$$
$$c3 \mathrel{\widehat{=}} b(i-1) - v < 0$$

monitored inputs: $cmd(i)$				$b(i)$	$d(i)$	$w(i)$	$r(i)$
$i = 0$				0	0	0	ok
$i > 0$	nothing			$b(i-1)$	$d(i-1)$	$w(i-1)$	$r(i-1)$
	dep(v)	$v > 0$		$b(i-1) + v$	$d(i-1) + v$	$w(i-1)$	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	c1		$b(i-1) - v$	$d(i-1)$	$w(i-1) + v$	ok
		$\neg$ c1	c2	NC	NC	NC	er1
			c3	NC	NC	NC	er2

Is the Function Table Disjoint?

$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i-1) - v \geq 0$

$c2 \mathrel{\widehat{=}} v \leq 0$

$c3 \mathrel{\widehat{=}} b(i-1) - v < 0$

For disjointness we must prove: $\neg(c2 \wedge c3)$!

LASSONDE SCHOOL OF ENGINEERING

Is the Function Table Disjoint?

$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i{-}1) - v \geq 0$

$c2 \mathrel{\widehat{=}} v \leq 0$

$c3 \mathrel{\widehat{=}} b(i{-}1) - v < 0$

For disjointness we must prove: $\neg(c2 \wedge c3)$!

$$\neg(c2 \wedge c3)$$

Is the Function Table Disjoint?

$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i-1) - v \geq 0$

$c2 \mathrel{\widehat{=}} v \leq 0$

$c3 \mathrel{\widehat{=}} b(i-1) - v < 0$

For disjointness we must prove: $\neg(c2 \wedge c3)$!

$$
\begin{array}{ll}
 & \neg(c2 \wedge c3) \\
\equiv & \quad \{\text{de Morgan}\} \\
 & \neg c2 \ \vee \ \neg c3
\end{array}
$$

Is the Function Table Disjoint?

$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i\text{–}1) - v \geq 0$

$c2 \mathrel{\widehat{=}} v \leq 0$

$c3 \mathrel{\widehat{=}} b(i\text{–}1) - v < 0$

For disjointness we must prove: $\neg(c2 \wedge c3)$!

$$
\begin{array}{ll}
 & \neg(c2 \wedge c3) \\
\equiv & \quad \{\text{de Morgan}\} \\
 & \neg c2 \ \vee \ \neg c3 \\
\equiv & \quad \{\text{defns of } c1 \text{ and } c2 \text{ and arithmetic}\} \\
 & v > 0 \ \vee \ b(i\text{–}1) - v \geq 0
\end{array}
$$

Is the Function Table Disjoint?

$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i\text{–}1) - v \geq 0$

$c2 \mathrel{\widehat{=}} v \leq 0$

$c3 \mathrel{\widehat{=}} b(i\text{–}1) - v < 0$

For disjointness we must prove: $\neg(c2 \wedge c3)$!

$$
\begin{array}{ll}
 & \neg(c2 \wedge c3) \\
\equiv & \quad \{\text{de Morgan}\} \\
 & \neg c2 \ \vee \ \neg c3 \\
\equiv & \quad \{\text{defns of } c1 \text{ and } c2 \text{ and arithmetic}\} \\
 & v > 0 \ \vee \ b(i\text{–}1) - v \geq 0 \\
\Leftarrow & \quad \{\text{arithmetic: } x \geq 0 \ \Rightarrow \ y > 0 \ \vee \ x - y \geq 0\} \\
 & b(i-1) \geq 0 \qquad (\mathrm{P})
\end{array}
$$

Is the Function Table Disjoint?

$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i\text{–}1) - v \geq 0$

$c2 \mathrel{\widehat{=}} v \leq 0$

$c3 \mathrel{\widehat{=}} b(i\text{–}1) - v < 0$

For disjointness we must prove: $\neg(c2 \wedge c3)$!

$$
\begin{array}{ll}
 & \neg(c2 \wedge c3) \\
\equiv & \quad \{\text{de Morgan}\} \\
 & \neg c2 \ \vee \ \neg c3 \\
\equiv & \quad \{\text{defns of } c1 \text{ and } c2 \text{ and arithmetic}\} \\
 & v > 0 \ \vee \ b(i\text{–}1) - v \geq 0 \\
\Leftarrow & \quad \{\text{arithmetic: } x \geq 0 \ \Rightarrow \ y > 0 \ \vee \ x - y \geq 0\} \\
 & b(i-1) \geq 0 \qquad (\mathrm{P})
\end{array}
$$

Note that we are using transitivity, i.e.

$y \leq 0 \leq x \Rightarrow x \geq y \ \equiv \ x \geq 0 \Rightarrow y > 0 \vee x - y \geq 0$

Is the Function Table Disjoint?

$c1 \mathrel{\widehat{=}} v > 0 \ \wedge \ b(i{-}1) - v \geq 0$

$c2 \mathrel{\widehat{=}} v \leq 0$

$c3 \mathrel{\widehat{=}} b(i{-}1) - v < 0$

For disjointness we must prove: $\neg(c2 \wedge c3)$!

$$
\begin{array}{ll}
 & \neg(c2 \wedge c3) \\
\equiv & \quad \{\text{de Morgan}\} \\
 & \neg c2 \ \vee \ \neg c3 \\
\equiv & \quad \{\text{defns of } c1 \text{ and } c2 \text{ and arithmetic}\} \\
 & v > 0 \ \vee \ b(i{-}1) - v \geq 0 \\
\Leftarrow & \quad \{\text{arithmetic: } x \geq 0 \ \Rightarrow \ y > 0 \ \vee \ x - y \geq 0\} \\
 & b(i-1) \geq 0 \qquad \text{(P)}
\end{array}
$$

Note that we are using transitivity, i.e.

$y \leq 0 \leq x \Rightarrow x \geq y \ \equiv \ x \geq 0 \Rightarrow y > 0 \vee x - y \geq 0$

So we need (P) to obtain disjointness and we don't have it!

What to do?

Corrected Function Table

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Notes

Corrected Function Table

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Notes

- The logical derivation on the previous slide allowed us to derive the disjointness condition: $b(i-1) \geq 0$

Corrected Function Table

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Notes

- The logical derivation on the previous slide allowed us to derive the disjointness condition: $b(i-1) \geq 0$
- We should thus be able to prove the invariant: $r(i) \neq logic\_err$

Corrected Function Table

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Notes

- The logical derivation on the previous slide allowed us to derive the disjointness condition: $b(i-1) \geq 0$
- We should thus be able to prove the invariant: $r(i) \neq logic\_err$
- If the program is implemented incorrectly (e.g. $2 * v$ is withdrawn insted of v) then the logical error handling will catch the problem at testing time

Corrected Function Table

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Notes

- Change: If we change the requirements to allow negative balances (with a credit limit) the logical error becomes an ok condition.

Corrected Function Table

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Notes

- Change: If we change the requirements to allow negative balances (with a credit limit) the logical error becomes an ok condition.
- The function table is updated and re-checked for completeness and disjointness.

Corrected Function Table

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Notes

- Change: If we change the requirements to allow negative balances (with a credit limit) the logical error becomes an ok condition.
- The function table is updated and re-checked for completeness and disjointness.
- If all that we had was the code, then we might forget to revise the error handling (in the error handler) to be consistent with the withdraw method in the model class.

Corrected Function Table

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Notes

- Change: If we change the requirements to allow negative balances (with a credit limit) the logical error becomes an ok condition.
- The function table is updated and re-checked for completeness and disjointness.
- If all that we had was the code, then we might forget to revise the error handling (in the error handler) to be consistent with the withdraw method in the model class.
- Note that an output $b(i-1)$ in the previous state is ok in the condition part of the function table (but not $b(i)$). Why?

Is the Function Table Disjoint?

Replace withdrawal with

```
withdraw(v):
      COND
           b(i-1) >= 0 AND v > 0 AND b(i-1) - v >= 0
        ->   withdraw?(v)(i) AND r(i) = ok,
           b(i-1) >= 0 AND v <= 0
        -> skip(i) AND r(i) = er1,
           b(i-1) >= 0 AND b(i-1) - v < 0
        -> skip(i) AND r(i) = er2,
           b(i-1) < 0
         -> r(i) = logic_err
      ENDCOND
```

Is the Function Table Disjoint?

Replace withdrawal with

```
withdraw(v):
      COND
           b(i-1) >= 0 AND v > 0 AND b(i-1) - v >= 0
        ->   withdraw?(v)(i) AND r(i) = ok,
           b(i-1) >= 0 AND v <= 0
        -> skip(i) AND r(i) = er1,
           b(i-1) >= 0 AND b(i-1) - v < 0
        -> skip(i) AND r(i) = er2,
           b(i-1) < 0
         -> r(i) = logic_err
      ENDCOND
```

New invariant to be proved

```
invariant3(i:DTIME): bool =
    r(i) /= logic_err

  bank_invariant1 : CONJECTURE
         (FORALL i: bank_ft(i) )
      =>
        FORALL i : invariant1(i)
```


LASSONDE SCHOOL OF ENGINEERING

Note: PVS Bug reported

The most embedded COND is not checked for disjointness

```
embedded_cond_bug: THEORY
BEGIN
 x,y,z: real
 b: nat

 f:bool =
   COND
     x < 0 -> b = 0,
     x >= 0 ->
       COND
      x = 0 -> b =1,
      x > 0 ->
         COND
           y <= 0 -> b = 2,
           y >= 0 -> b = 3
         ENDCOND
    ENDCOND
   ENDCOND
END embedded_cond_bug
```


LASSONDE SCHOOL OF ENGINEERING

Is the Function Table Disjoint?

```
withdraw(v):
      COND
           b(i-1) >= 0 AND v > 0 AND b(i-1) - v >= 0
        ->   withdraw?(v)(i) AND r(i) = ok,
           b(i-1) >= 0 AND v <= 0
        -> skip(i) AND r(i) = er1,
           b(i-1) >= 0 AND b(i-1) -v < 0
        -> skip(i) AND r(i) = er2,
           b(i-1) < 0
         -> r(i) = logic_err
      ENDCOND
```

Case b(i-1) < 0?

Given that we can prove that it never happens, why did we need PVS to signal a TCC violation?

Is the Function Table Disjoint?

```
withdraw(v):
      COND
           b(i-1) >= 0 AND v > 0 AND b(i-1) - v >= 0
        ->   withdraw?(v)(i) AND r(i) = ok,
           b(i-1) >= 0 AND v <= 0
        -> skip(i) AND r(i) = er1,
           b(i-1) >= 0 AND b(i-1) -v < 0
        -> skip(i) AND r(i) = er2,
           b(i-1) < 0
         -> r(i) = logic_err
      ENDCOND
```

Case b(i-1) < 0?

Given that we can prove that it never happens, why did we need PVS to signal a TCC violation?

Suppose we introduce a change: allow balances to go negative up to a credit limit?

Is the Function Table Disjoint?

```
withdraw(v):
      COND
           b(i-1) >= 0 AND v > 0 AND b(i-1) - v >= 0
        ->   withdraw?(v)(i) AND r(i) = ok,
           b(i-1) >= 0 AND v <= 0
        -> skip(i) AND r(i) = er1,
           b(i-1) >= 0 AND b(i-1) -v < 0
        -> skip(i) AND r(i) = er2,
           b(i-1) < 0
         -> r(i) = logic_err
      ENDCOND
```

Case b(i-1) < 0?

Given that we can prove that it never happens, why did we need PVS to signal a TCC violation?

Suppose we introduce a change: allow balances to go negative up to a credit limit?

In our code we would change "withdraw?", but might forget to eliminate the error handling and a spurious error would result.

Function Table Conciseness/Precision

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Function Table Conciseness/Precision

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Conciseness

- How long would it take to write out the specification, informally, in English?
- Would it be vague? Or would it have the precision of the function table?
- Would the informal English be complete and disjoint?

Function Table Conciseness/Precision

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Conciseness

- How long would it take to write out the specification, informally, in English?
- Would it be vague? Or would it have the precision of the function table?
- Would the informal English be complete and disjoint?
- Recall something as simple as leap-year?

Validation of invariants and use cases

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Validation of invariants and use cases

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Validation

- Can we use the function table to prove R3, viz, that $\forall i \in DTIME : balance(i) \geq 0$?

Validation of invariants and use cases

monitored inputs $cmd(i)$				b(i)	d(i)	w(i)	r(i)
$i = 0$				0	0	0	ok
$i > 0$	nothing			b(i -1)	NC	NC	NC
	dep(v)	$v > 0$		b(i -1) + v	d(i -1) + v	NC	ok
		$v \leq 0$		NC	NC	NC	er1
	with(v)	$b(i-1) \geq 0$	$v > 0 \wedge b(i-1) - v \geq 0$	b(i -1) - v	d(i -1)	w(i -1) + v	ok
			$v < 0$	NC	NC	NC	er1
			$b(i-1) - v < 0$	NC	NC	NC	er2
		$b(i-1) < 0$		NC	NC	NC	logic_err
Assume: $v \in \mathbb{R}$							

Validation

- Can we use the function table to prove R3, viz, that $\forall i \in DTIME : balance(i) \geq 0$?
- Can we use the function table to show that a Use Case is satisfied?
- How?

Time Theory

Time Theory has parameter *delta*, the sampling time.

Time Theory

Time Theory has parameter *delta*, the sampling time.

```
Time[delta: posreal]   : THEORY
BEGIN
  % digital time
  DTIME : TYPE = nat

  % psuedo, digitized real time
  RTIME : TYPE =
    {t:nnreal| (EXISTS (i:DTIME) : t = i * delta)}

  % actual time
  TIME : TYPE = nnreal

  % Positive DTIME
  POS_DTIME: TYPE = posnat

  ...other functions

END Time
```

Bank Theory

```
SmallBank[delta:posreal]: THEORY
BEGIN
  IMPORTING Time[delta]
  i: VAR DTIME

  STATUS    : TYPE = {ok, er1, er2}
% Inputs
  v: real

  COMMAND: DATATYPE
    BEGIN
      nothing             : nothing?
      deposit   (x:real)  : deposit?
      withdraw  (x:real)  : withdraw?
    END COMMAND

  cmd: [DTIME -> COMMAND]

% Outputs
  ...
```


LASSONDE SCHOOL OF ENGINEERING

Bank Theory

```
SmallBank[delta:posreal]: THEORY
BEGIN
  IMPORTING Time[delta]
  i: VAR DTIME

  STATUS    : TYPE = {ok, er1, er2}
% Inputs
  v: real

  COMMAND: DATATYPE
    BEGIN
      nothing             : nothing?
      deposit   (x:real)  : deposit?
      withdraw  (x:real)  : withdraw?
    END COMMAND

  cmd: [DTIME -> COMMAND]

% Outputs
  ...
```

Note that formatting is important!

LASSONDE SCHOOL OF ENGINEERING

Monitored Events as a DATATYPE

```
...
 COMMAND: DATATYPE
   BEGIN
     nothing             : nothing?
     deposit (x:real)    : deposit?
     withdraw (x:real)   : withdraw?
   END COMMAND

 i: VAR DTIME
 cmd: VAR [DTIME -> COMMAND]
  ..
```

Commands

- The commands are constructors
- Can say: cmd(i) = deposit(v) AND b(i) = 425.00 ...

See PVS Language Reference

Monitored Events as a DATATYPE

An example of a datatype is stack:

```
stack[T: TYPE]: DATATYPE
BEGIN
empty: empty?
push(top:T, pop:stack): nonempty?
END stack
```

The stack datatype has two constructors, empty and push, that allow stack elements to be constructed. For example, the term push(1, empty) is an element of type stack[int].

The recognizers empty? and nonempty? are predicates over the stack datatype that are true when their argument is constructed using the corresponding constructor. Given a stack element that is known to be nonempty?, the accessors top and pop may be used to extract the first and second arguments.

See PVS Language Reference p72

Outputs and Queries

```
% Outputs
  b:  [DTIME -> real]     % balance
  d:  [DTIME -> real]     % total deposits
  w:  [DTIME -> real]     % total withdraws
  r:  [DTIME -> STATUS]   % report

% Queries
  deposit?(x: real)(i:POS_DTIME): bool =
     b(i) = b(i-1) + x AND d(i) = d(i-1) + x
                       AND w(i) = w(i-1)

  withdraw?(x: real)(i:POS_DTIME): bool =
     b(i) = b(i-1) - x AND d(i) = d(i-1)
                       AND w(i) = w(i-1) + x

  skip(i:POS_DTIME): bool =
    b(i) = b(i-1) AND d(i) = d(i-1)
                  AND w(i) = w(i-1)
  init: bool =
    b(0) = 0 AND d(0) = 0 AND w(0) = 0
```

PVS Function Table 1

```
% Response Function Table
  bank_ft (i): bool =
    COND
      i = 0 ->  init AND r(0) = ok,
      i > 0 ->
        CASES cmd(i) OF
          nothing    : skip(i) AND r(i) = r(i-1),
          deposit(v):
            COND
              v > 0 -> deposit?(v)(i) AND r(i) = ok,
              ELSE  -> skip(i)        AND r(i) = er1
            ENDCOND,
          withdraw(v): ...
        ENDCASES
    ENDCOND
```

PVS Function Table 1

```
% Response Function Table
  bank_ft (i): bool =
    COND
      i = 0 ->  init AND r(0) = ok,
      i > 0 ->
        CASES cmd(i) OF
          nothing    : skip(i) AND r(i) = r(i-1),
          deposit(v):
            COND
              v > 0 -> deposit?(v)(i) AND r(i) = ok,
              ELSE  -> skip(i)        AND r(i) = er1
            ENDCOND,
          withdraw(v): ...
        ENDCASES
    ENDCOND
```

Do not confuse deposit(v) as a command and deposit?(x) as a query

Use CASES for the commands (datatype).

PVS Function Table 2

```
% Response Function Table
  COND ...
       i > 0 ->
         CASES cmd(i) OF
           nothing    :  ...
           deposit(v) : ...
           withdraw(v):
             COND
                  v > 0 AND b(i-1) - v >= 0
               ->       withdraw?(v)(i) AND r(i) = ok,
                 NOT ( v > 0 AND b(i-1) - v >= 0)
               ->   COND
                      v <= 0
                    -> skip(i) AND r(i) = er1,
                     b(i-1) -v < 0
                    -> skip(i) AND r(i) = er2
                    ENDCOND
             ENDCOND
         ENDCASES
    ENDCOND
```

PVS Function Table with Naming Abstraction

```
bank_ft (i): bool =
    COND
      i = 0 ->  init AND r(0) = ok,
      i > 0 ->
        CASES cmd(i) OF
          ...
           withdraw(v):
            COND
                 b(i-1) >= 0 AND v > 0 AND b(i-1) - v >= 0
              ->      withdraw?(v)(i) AND r(i) = ok,
                 b(i-1) >= 0 AND v <= 0
              -> skip(i) AND r(i) = er1,
                 b(i-1) >= 0 AND b(i-1) -v < 0
              -> skip(i) AND r(i) = er2,
                 b(i-1) < 0
               -> r(i) = logic_err
            ENDCOND
        ENDCASES
    ENDCOND
```

PVS Function Table

So what can we do with all this?

Validation

How?

- Invariants such as R3
- Use Cases

Validate invariant R3

```
% Invariants
  invariant1(i:DTIME): bool =
    0 <= b(i)

  bank_invariant1 : CONJECTURE
        (FORALL i: bank_ft(i) )
      =>
        FORALL i : invariant1(i)
```

Validate invariant R3

```
% Invariants
  invariant1(i:DTIME): bool =
    0 <= b(i)

  bank_invariant1 : CONJECTURE
        (FORALL i: bank_ft(i) )
      =>
        FORALL i : invariant1(i)
```

```
bank_invariant1 :

{-1}   (FORALL i: bank_ft(i))
   |-------
{1}    FORALL i: invariant1(i)

Rule?
```

What rule to apply?

Validate invariant R3

```
  invariant1(b,d,w)(i:DTIME): bool = 0 <= b(i)

  bank_invariant1 : CONJECTURE
        (FORALL i: bank_ft1(cmd, b, d, w, r)(i))
      =>
        FORALL i : invariant1(b, d, w)(i)
```

```
{-1}   FORALL i: bank_ft1(cmd, b, d, w, r)(i)
   |-------
{1}    FORALL i: invariant2(b, d, w)(i)

Rule?
```

What rule to apply?

Induction

Validate invariant R3: induct results in 2 subgoals

```
 {-1}   (FORALL i: bank_ft(i))
  |-------
{1}    FORALL i: invariant1(i)

Rule? (induct "i")
Inducting on i on formula 1, yields  2 subgoals:
bank_invariant1.1 :

[-1]   (FORALL i: bank_ft(i))
  |-------
{1}    invariant1(0)
Rule?
```

```
bank_invariant1.2 :

[-1]   (FORALL i: bank_ft(i))
  |-------
{1}    FORALL j:
            invariant1(j) IMPLIES invariant1(j + 1)
Rule?
```

Validate invariant R3: first sub-goal base case $i = 0$

```
bank_invariant1.1 :
[-1]  FORALL i: bank_ft(i)
   |-------
{1}    invariant1(0)
Rule? (inst -1 "0")

Instantiating the top quantifier.... simplifies to:
bank_invariant1.1 :
{-1}  bank_ft(0)
   |-------
   [1]    invariant(0)
Rule? (grind)

invariant1 rewrites invariant1(0)
  to 0 <= b(0)
Trying repeated skolemization, ...
This completes the proof of bank_invariant1.1.

bank_invariant1.2 :  ....
```

Validate invariant R3: second sub-goal inductive case

```
bank_invariant1.2 :
[-1]   FORALL i: bank_ft(i)
   |-------
{1}    FORALL j: invariant1(j)
             IMPLIES invariant1(j + 1)
Rule? (skeep)
Skolemizing...simplifies to:
bank_invariant1.2 :

{-1}   invariant1(j)
[-2]   FORALL i: bank_ft(i)
   |-------
{1}    invariant1(j + 1)
Rule?
```

What rule now?

Validate invariant R3: second sub-goal inductive case

```
bank_invariant1.2 :
{-1}  invariant1(j)
[-2]  FORALL i: bank_ft(i)
  |-------
{1}   invariant1(j + 1)
Rule? (inst -2 "j+1")
Instantiating...simplifies to:
bank_invariant1.2 :
[-1]  invariant1(j)
{-2}  bank_ft(j + 1)
  |-------
[1]   invariant1(j + 1)
Rule? (grind)
invariant1 rewrites ....
Trying repeated skolemization, ...
This completes the proof of bank_invariant1.2.
Q.E.D
```


LASSONDE SCHOOL OF ENGINEERING

Validate invariant R3: M-x show-proof

```
;;; Proof bank_invariant1-1
;;;for formula SmallBank.bank_invariant1
;;; developed with shostak decision procedures
(""
 (skeep)
 (induct "i")
 (("1" (inst -1 "0") (grind))
  ("2" (skeep) (inst -2 "j+1") (grind))))
```

We can prove a stronger invariant

```
invariant2(i:DTIME): bool =
        0 <= b(i)
   AND  0 <= d(i)
   AND  0 <= w(i)
   AND  d(i) - w(i) = b(i)

 bank_invariant2 : CONJECTURE
        (FORALL i: bank_ft(i))
     =>
       FORALL i : invariant2(i)
```

Proving an abstract Use Case 1

```
use_case1_v1: CONJECTURE
          i > 0
      AND b(i) = 10.25
      AND bank_ft(i+1)
      AND bank_ft(i+2)
      AND cmd(i+1) = deposit (4.75)
      AND cmd(i+2) = withdraw(10.00)
      =>
          b(i+2) = 5.00
```

- What proof rule?

Proving an abstract Use Case 1

```
use_case1_v1: CONJECTURE
          i > 0
      AND b(i) = 10.25
      AND bank_ft(i+1)
      AND bank_ft(i+2)
      AND cmd(i+1) = deposit (4.75)
      AND cmd(i+2) = withdraw(10.00)
      =>
          b(i+2) = 5.00
```

- What proof rule?
- Try grind

Proving an abstract Use Case 2

```
% Need to figure out what XXXX is?
% Use the method of weakest preconditions
  x,y: posreal
  use_case2: CONJECTURE
           i > 0
      AND XXXX
      AND bank_ft1(i+1)
      AND bank_ft1(i+2)
      AND cmd(i+1) = deposit (x)
      AND cmd(i+2) = withdraw(y)
      =>
          b(i+2) = b(i) + x - y
      AND d(i+2) = d(i) + x
      AND w(i+2) = w(i) + y
```

- Work out what XXX is using weakest preconditions

Proving an abstract Use Case 2

```
% Need to figure out what XXXX is?
% Use the method of weakest preconditions
  x,y: posreal
  use_case2: CONJECTURE
           i > 0
      AND XXXX
      AND bank_ft1(i+1)
      AND bank_ft1(i+2)
      AND cmd(i+1) = deposit (x)
      AND cmd(i+2) = withdraw(y)
      =>
          b(i+2) = b(i) + x - y
      AND d(i+2) = d(i) + x
      AND w(i+2) = w(i) + y
```

- Work out what XXX is using weakest preconditions
- What proof rule?

Proving an abstract Use Case 2

```
% Need to figure out what XXXX is?
% Use the method of weakest preconditions
  x,y: posreal
  use_case2: CONJECTURE
           i > 0
      AND XXXX
      AND bank_ft1(i+1)
      AND bank_ft1(i+2)
      AND cmd(i+1) = deposit (x)
      AND cmd(i+2) = withdraw(y)
      =>
          b(i+2) = b(i) + x - y
      AND d(i+2) = d(i) + x
      AND w(i+2) = w(i) + y
```

- Work out what XXX is using weakest preconditions
- What proof rule?
- Try grind

Using a RECORD for the state

State encoded as a record type and initial state

```
STATE: TYPE = [# b: real
              , d: real
              , w: real
              , r: STATUS
              #]

st: [DTIME -> STATE]

init_state: STATE =
  (# b := 0, d := 0, w:=0, r := ok #)
```

System I/O Specification

```
deposit_ft(x: real)(i:POS_DTIME): bool =
  COND
    x > 0 -> st(i) = st(i-1) WITH [ b := b_ + x
                                  ,d := d_ + x
                                  ,r := ok
                                 ],
    x <= 0  -> st(i) = st(i-1) WITH [r := er1]
  ENDCOND
    WHERE  b_ = st(i-1)`b
          ,d_ = st(i-1)`d
```


LASSONDE SCHOOL OF ENGINEERING

System I/O Specification

```
deposit_ft(x: real)(i:POS_DTIME): bool =
  COND
    x > 0 -> st(i) = st(i-1) WITH [ b := b_ + x
                                  ,d := d_ + x
                                  ,r := ok
                                 ],
    x <= 0  -> st(i) = st(i-1) WITH [r := er1]
  ENDCOND
    WHERE  b_ = st(i-1)`b
          ,d_ = st(i-1)`d
```

```
bank_ft (i): bool =
  COND
    i = 0 ->  st(0) = init_state,
    i > 0 ->
      CASES cmd(i) OF
        nothing     : st(i) = st(i-1),
        deposit(v)  : deposit_ft(v)(i),
        withdraw(v): withdraw_ft(v)(i)
      ENDCASES
  ENDCOND
```


LASSONDE SCHOOL OF ENGINEERING

Validation of Invariant Requirement

```
bank_ft (i): bool =
  COND
    i = 0 ->  st(0) = init_state,
    i > 0 ->
      CASES cmd(i) OF
        nothing    : st(i) = st(i-1),
        deposit(v) : deposit_ft(v)(i),
        withdraw(v): withdraw_ft(v)(i)
      ENDCASES
  ENDCOND
```

```
invariant1(i:DTIME): bool =
    0 <= st(i)`b
```

Validation of Invariant Requirement

```
bank_ft (i): bool =
  COND
    i = 0 ->  st(0) = init_state,
    i > 0 ->
      CASES cmd(i) OF
        nothing    : st(i) = st(i-1),
        deposit(v) : deposit_ft(v)(i),
        withdraw(v): withdraw_ft(v)(i)
      ENDCASES
  ENDCOND
```

```
invariant1(i:DTIME): bool =
    0 <= st(i)`b
```

```
bank_invariant1 : CONJECTURE
       (FORALL i: bank_ft(i) )
     =>
       FORALL i : invariant1(i)
```

Validation of Use Case Requirement

```
bank_ft (i): bool =
  COND
    i = 0 ->  st(0) = init_state,
    i > 0 ->
      CASES cmd(i) OF
        nothing    : st(i) = st(i-1),
        deposit(v) : deposit_ft(v)(i),
        withdraw(v): withdraw_ft(v)(i)
      ENDCASES
  ENDCOND
```

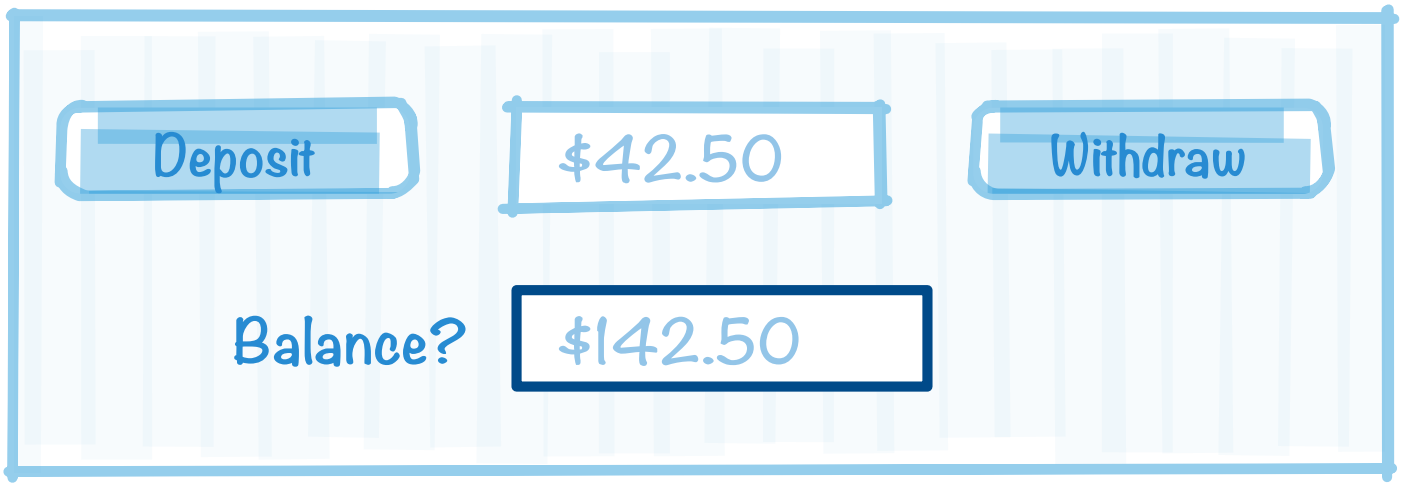
Validation of Use Case Requirement

```
bank_ft (i): bool =
  COND
    i = 0 ->  st(0) = init_state,
    i > 0 ->
      CASES cmd(i) OF
        nothing    : st(i) = st(i-1),
        deposit(v) : deposit_ft(v)(i),
        withdraw(v): withdraw_ft(v)(i)
      ENDCASES
  ENDCOND
```

```
% Proof this without grind
  use_case1: CONJECTURE
          i > 0
      AND st(i)`b = 10.25
      AND bank_ft(i+1)
      AND bank_ft(i+2)
      AND cmd(i+1) = deposit (4.75)
      AND cmd(i+2) = withdraw(10.00)
      =>
          st(i+2)`b = 5.00
```


LASSONDE SCHOOL OF ENGINEERING

Informal Bank ATM Requirements

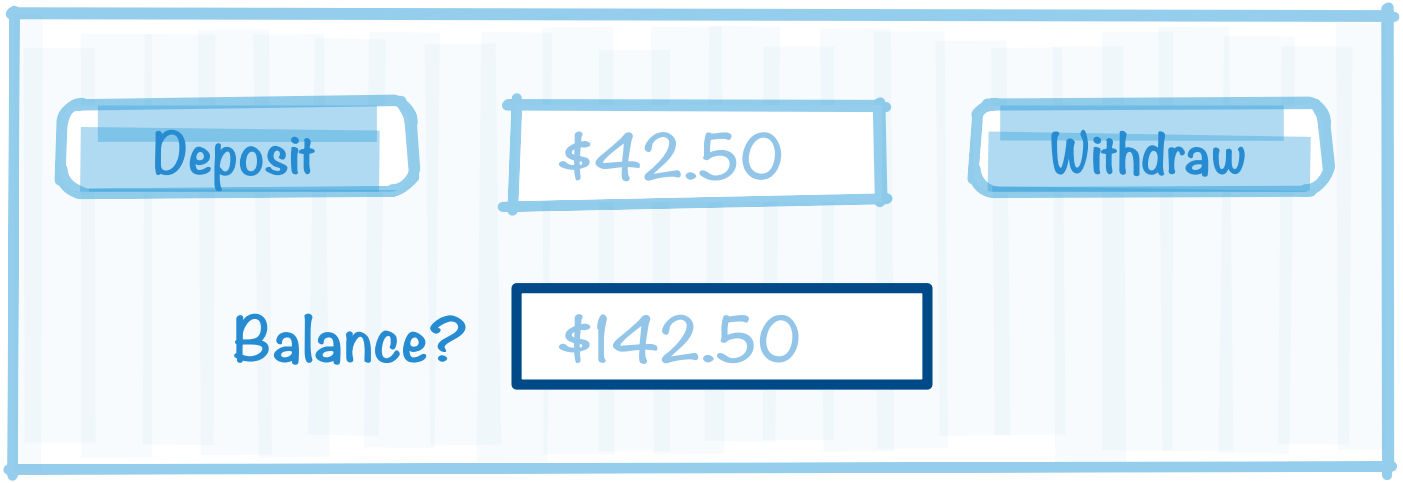


- R1 A user shall be able to deposit and withdraw money at the ATM
- R2 The current balance, total deposits and total withdrawals are displayed at the ATM
- R3 The balance shall never be negative

Compared with:

LASSONDE SCHOOL OF ENGINEERING

Informal Bank ATM Requirements



- R1 A user shall be able to deposit and withdraw money at the ATM
- R2 The current balance, total deposits and total withdrawals are displayed at the ATM
- R3 The balance shall never be negative

Compared with:

The formal requirements document as a function table

With the formal document we can predict/calculate that it is safe and fit for use.

Informal vs. Precise Requirements Document

Toyota Crash

"Runaway Toyota cases ignored"

LA Times 2009

By RALPH VARTABEDIAN AND KEN BENSINGER

NOVEMBER 8, 2009

Los Angeles Times

More than 1,000 Toyota and Lexus owners have reported since 2001 that their vehicles suddenly accelerated on their own, in many cases slamming into trees, parked cars and brick walls, among other obstacles, a Times review of federal records has found.

The crashes resulted in at least 19 deaths and scores of injuries over the last decade, records show. Federal regulators say that is far more than any other automaker has experienced.

Owner complaints helped trigger at least eight investigations into sudden acceleration in Toyota and Lexus vehicles by the National Highway Traffic Safety Administration in the last seven years. Toyota Motor Corp. recalled fewer than 85,000 vehicles in response to two of those probes, and the federal agency closed six other cases without finding a defect.

LASSONDE SCHOOL OF ENGINEERING

Informal vs. Precise Requirements Document

Toyota Crash

"Runaway Toyota cases ignored"

LA Times 2009

By RALPH VARTABEDIAN AND KEN BENSINGER

NOVEMBER 8, 2009

Los Angeles Times

More than 1,000 Toyota and Lexus owners have reported since 2001 that their vehicles suddenly accelerated on their own, in many cases slamming into trees, parked cars and brick walls, among other obstacles, a Times review of federal records has found.

The crashes resulted in at least 19 deaths and scores of injuries over the last decade, records show. Federal regulators say that is far more than any other automaker has experienced.

Owner complaints helped trigger at least eight investigations into sudden acceleration in Toyota and Lexus vehicles by the National Highway Traffic Safety Administration in the last seven years. Toyota Motor Corp. recalled fewer than 85,000 vehicles in response to two of those probes, and the federal agency closed six other cases without finding a defect.

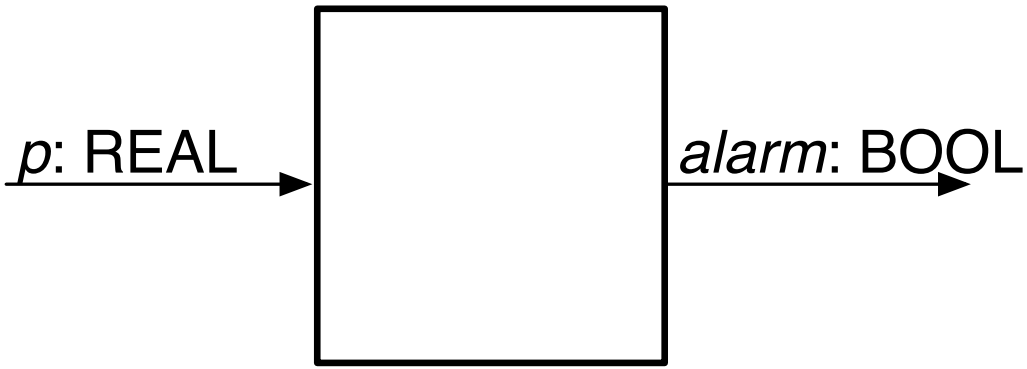
Precise

With a precise document we can predict/calculate that it is safe and fit for use!

LASSONDE SCHOOL OF ENGINEERING

Specification of the Pressure example

Consider a device with a monitored variable pressure $p : \mathbb{R}$ and controlled variable *alarm* as the output as shown in the figure.

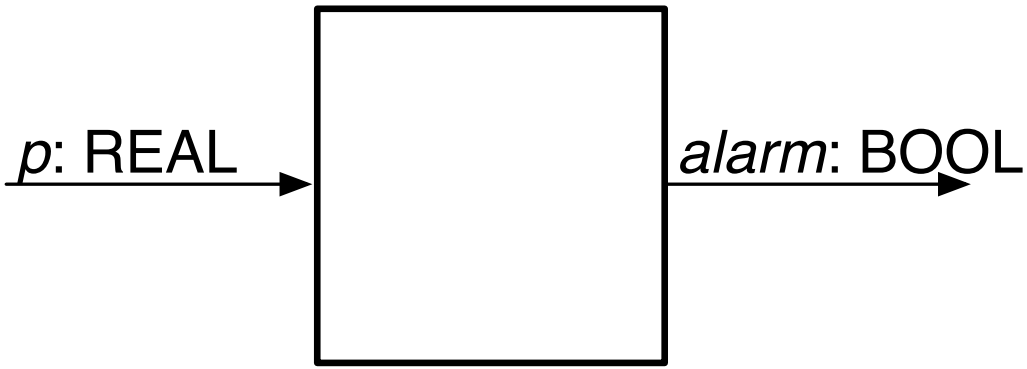


	alarm
$p \leq normal$	*False*
$normal < p < hi$	No Change
$p \geq hi$	*True*

LASSONDE SCHOOL OF ENGINEERING

Specification of the Pressure example

Consider a device with a monitored variable pressure $p : \mathbb{R}$ and controlled variable *alarm* as the output as shown in the figure.



	alarm
$p \leq normal$	*False*
$normal < p < hi$	No Change
$p \geq hi$	*True*

PVS Time Theory?

How do we specify the pressure example in T-ASM theory?

Specification of the Pressure example

```
alert: THEORY
BEGIN
  delta: posreal = 1 % TR = 1 seconds
  normal, hi: real
  IMPORTING Time[delta]
  p:      [DTIME -> real]   % Pressure
  alarm: [DTIME -> bool]

  spec(i:DTIME): bool =
    COND
        i = 0 -> NOT alarm(0)
      , i > 0 ->
        COND
           p(i) <= normal   -> alarm(i)=FALSE
         ,normal < p(i) AND p(i) < hi ->
                              alarm(i) = alarm(i-1)
         ,p(i) >= hi        -> alarm(i) = TRUE
         ENDCOND
      ENDCOND
```

Specification of the Pressure example

```
alert: THEORY
BEGIN
  delta: posreal = 1 % TR = 1 seconds
  normal, hi: real
  IMPORTING Time[delta]
  p:      [DTIME -> real]   % Pressure
  alarm: [DTIME -> bool]

  spec(i:DTIME): bool =
    COND
        i = 0 -> NOT alarm(0)
      , i > 0 ->
        COND
           p(i) <= normal   -> alarm(i)=FALSE
         ,normal < p(i) AND p(i) < hi ->
                              alarm(i) = alarm(i-1)
         ,p(i) >= hi        -> alarm(i) = TRUE
         ENDCOND
      ENDCOND
```

"spec" does not typecheck. Why?

Specification of the Pressure example

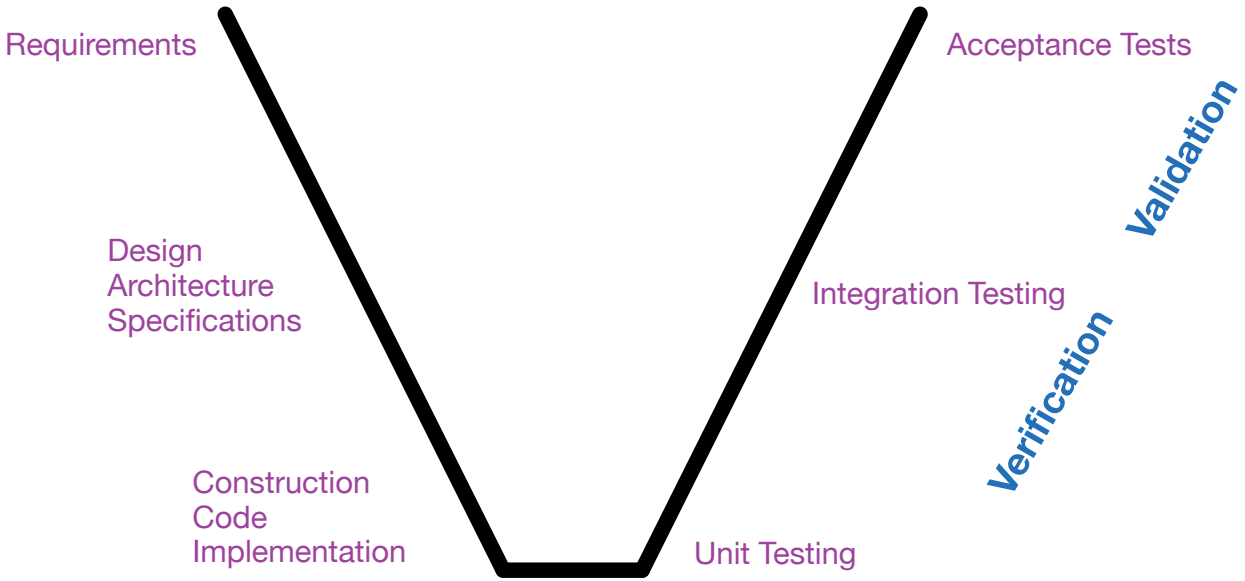
```
alert: THEORY
BEGIN
  delta: posreal = 1 % TR = 1 seconds
  normal, hi: real
  IMPORTING Time[delta]
  p:      [DTIME -> real]   % Pressure
  alarm: [DTIME -> bool]

  spec(i:DTIME): bool =
    COND
        i = 0 -> NOT alarm(0)
      , i > 0 ->  COND
           p(i) <= normal  -> alarm(i)=FALSE
          ,normal < p(i) AND p(i) < hi ->
                              alarm(i) = alarm(i-1)
          ,p(i) >= hi       -> alarm(i) = TRUE
          ENDCOND
    ENDCOND
```

ax: AXIOM normal < hi

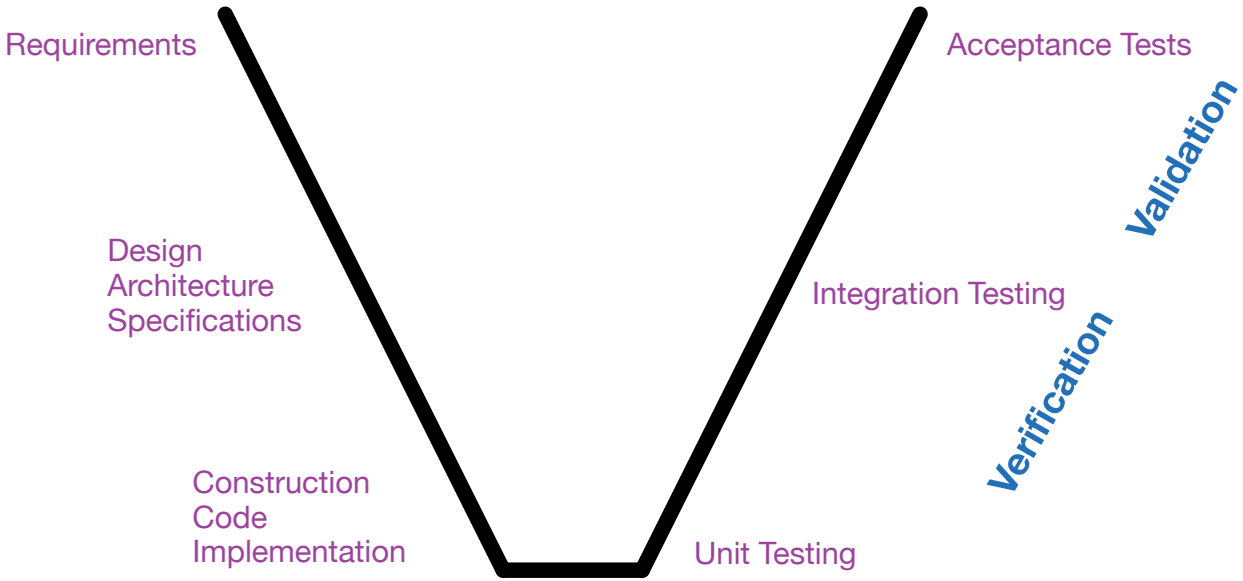
LASSONDE SCHOOL OF ENGINEERING

V&V curve



LASSONDE SCHOOL OF ENGINEERING

V&V curve

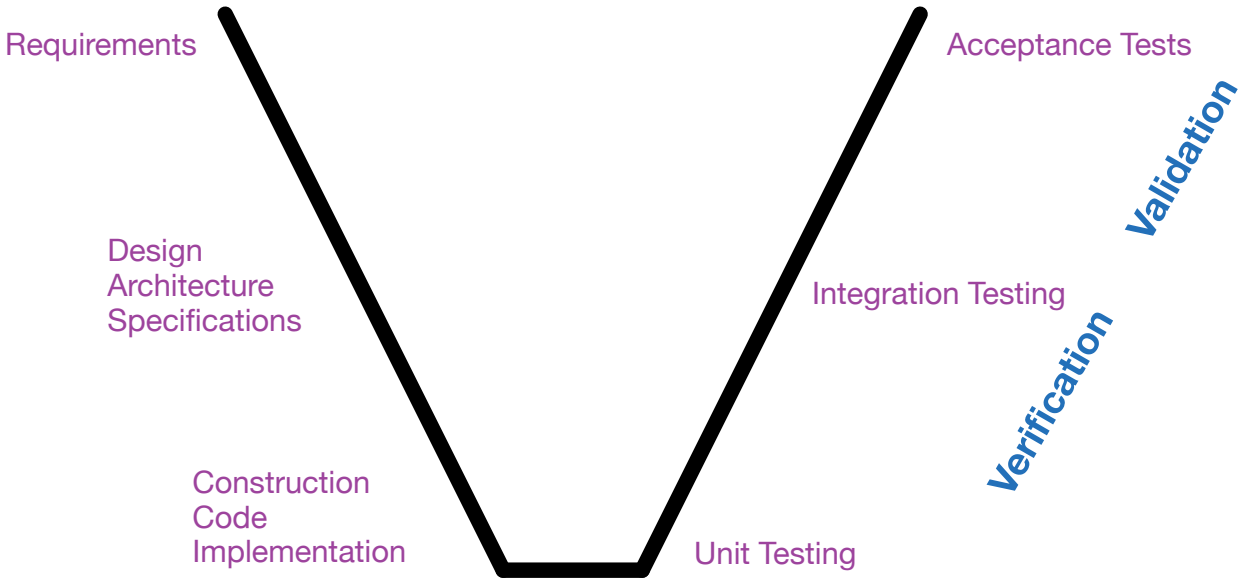


Validation

- Timed ASM Theory can be used to specify continuous as well as discrete systems

LASSONDE SCHOOL OF ENGINEERING

V&V curve



Validation

- Timed ASM Theory can be used to specify continuous as well as discrete systems
- Validation means checking that requirements are complete, disjoint and well-defined
- Validation means checking that requirements satisfy (a) Safety Invariants and (b) Use Cases (generalizations of Acceptance Tests)