Embedded Software: Better Models, Better Code

Tom Henzinger

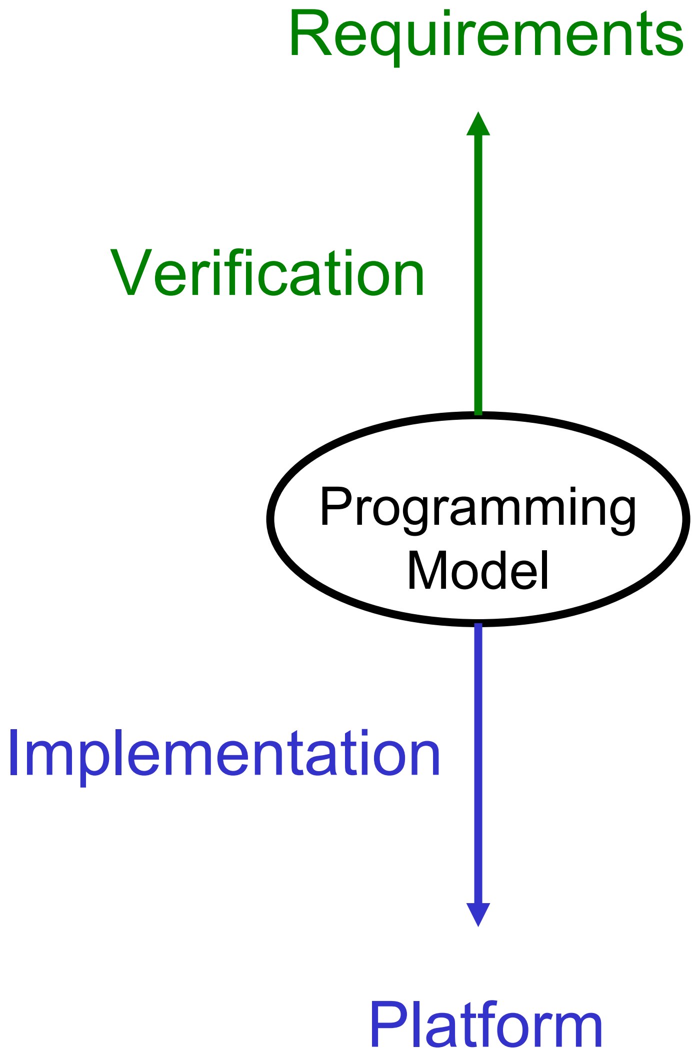
University of California, Berkeley

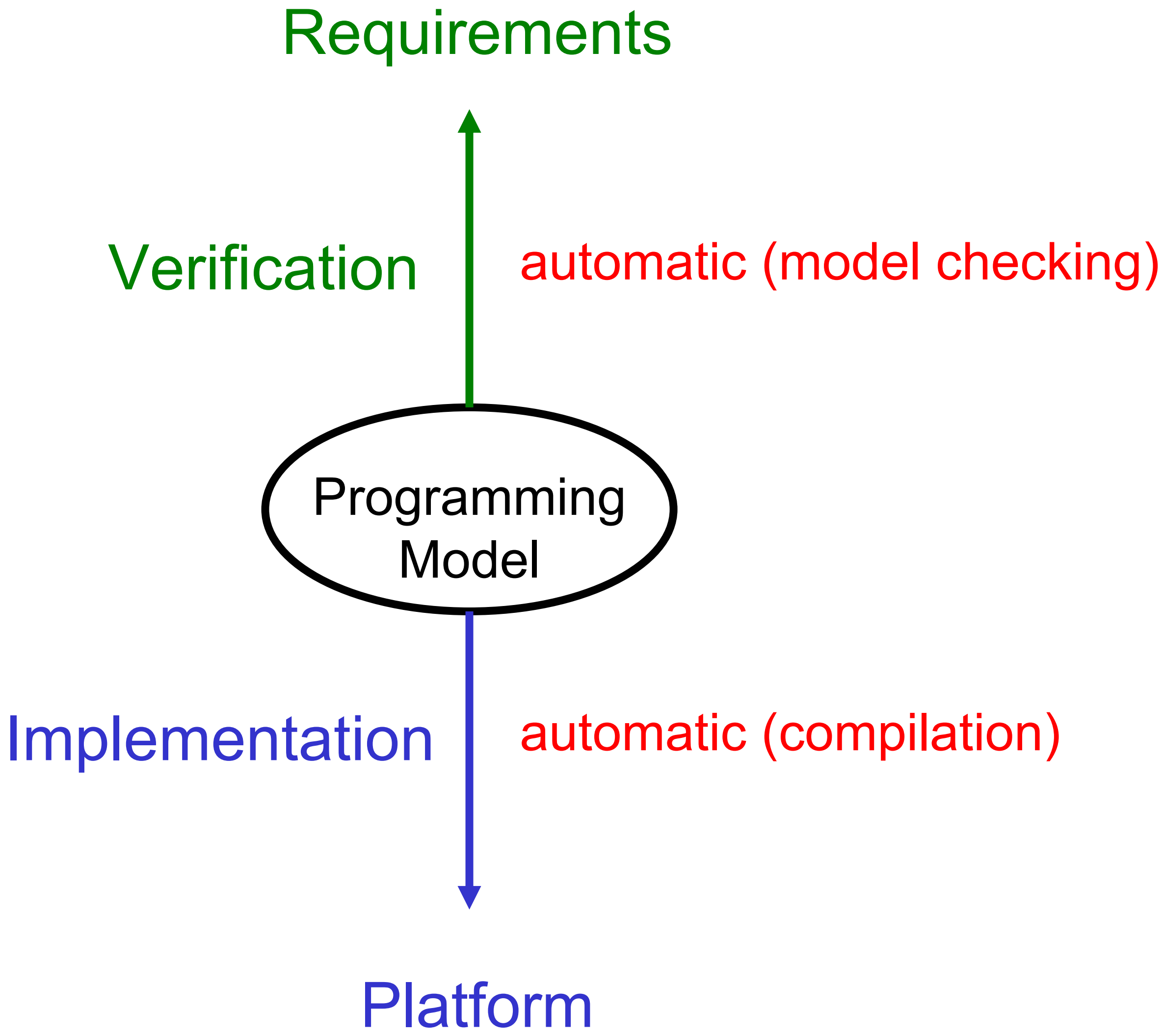
Our Research Explores Three Paradigms

In modeling, use discounted quantitative measures.

In composition, treat inputs and outputs contra-variantly.

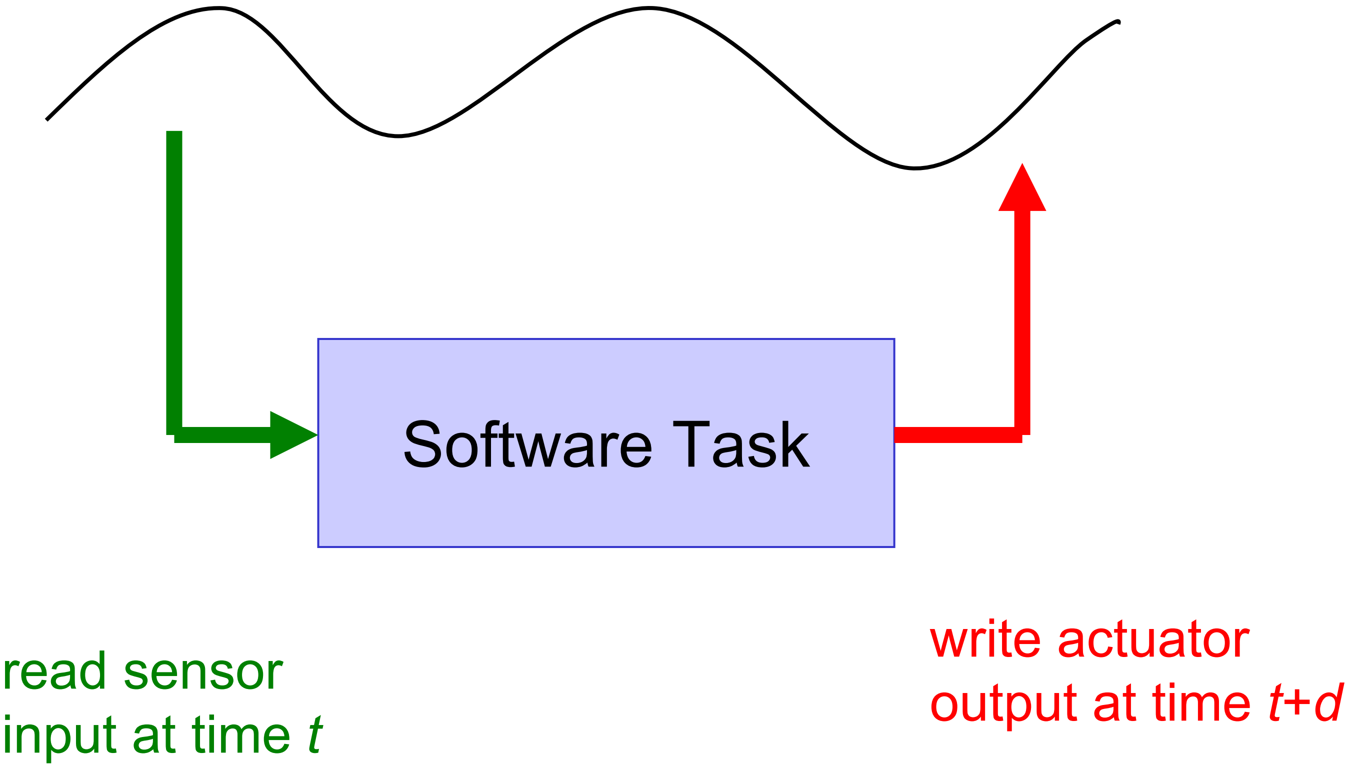
In implementation, preserve logical execution times.


Requirements
Verification
Programming Model
Implementation
Platform

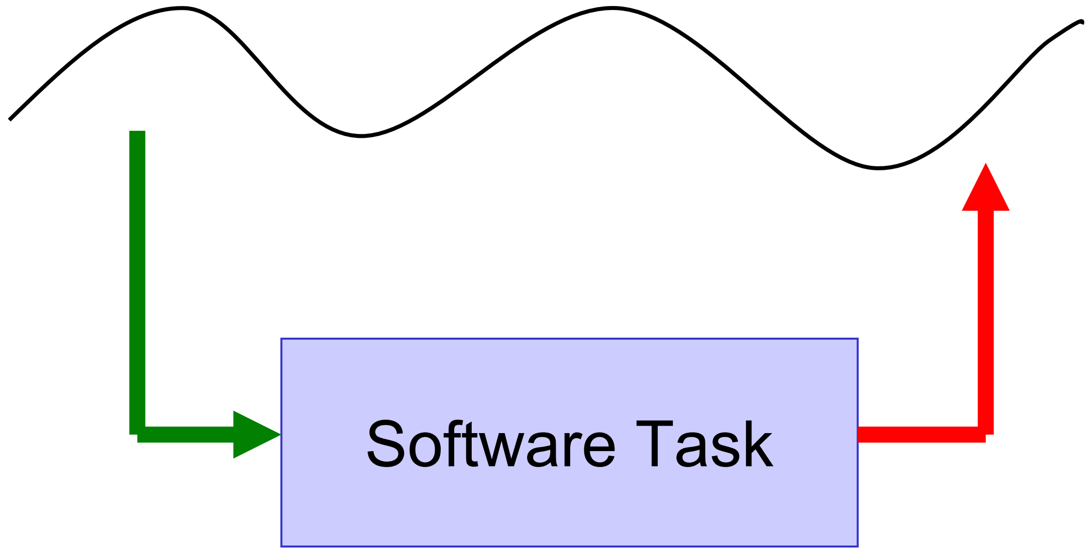


Requirements
Verification
automatic (model checking)
Programming Model
Implementation
automatic (compilation)
Platform

The LET (Logical Execution Time) Assumption



The LET (Logical Execution Time) Assumption

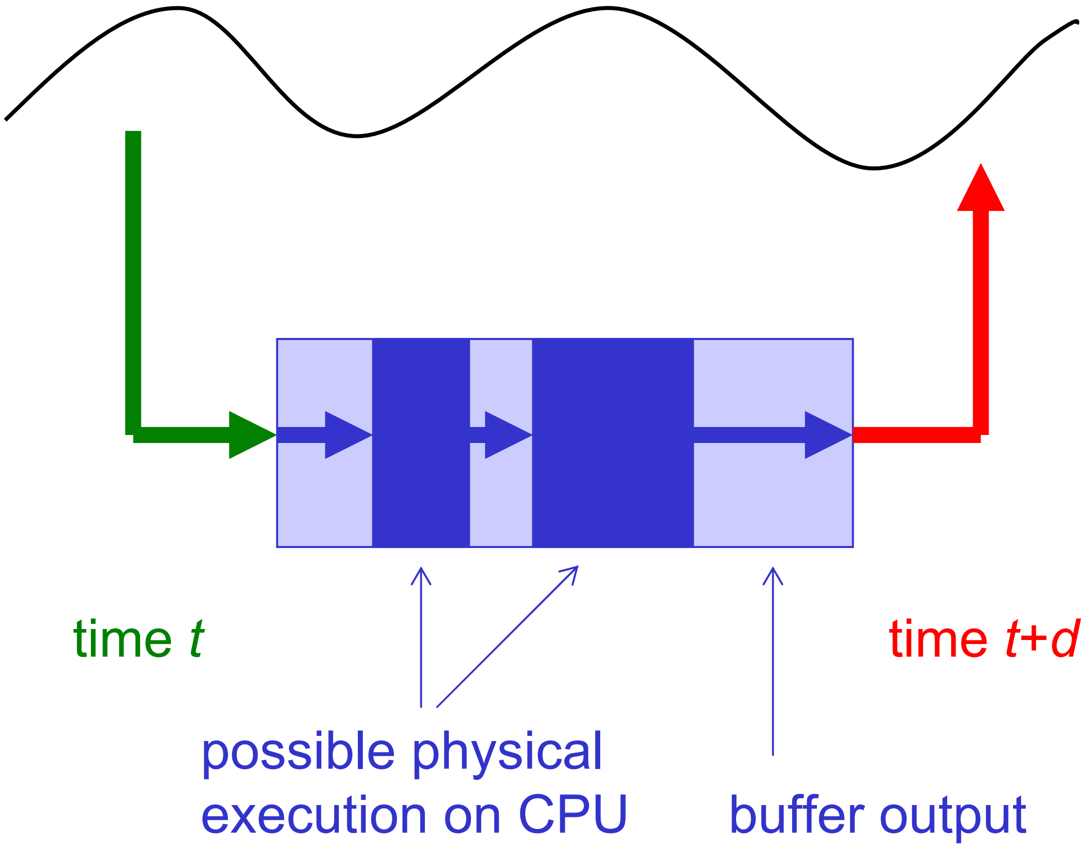


read sensor
input at time *t*

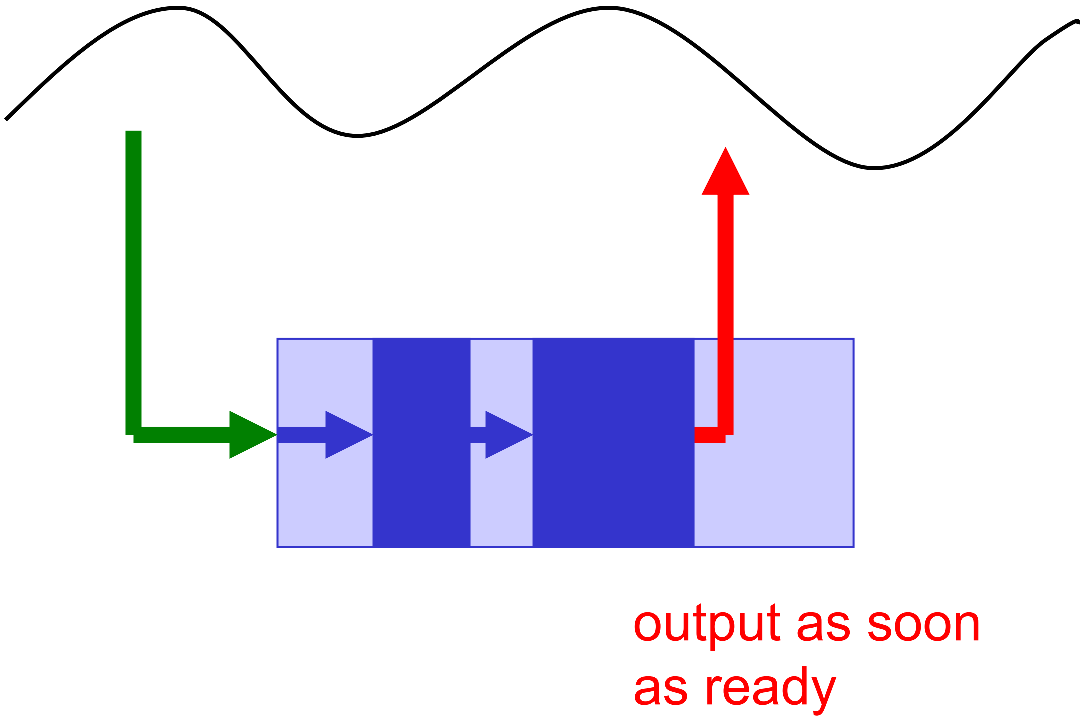
d>0 is the
task's "logical
execution time"

write actuator
output at time *t*+*d*,
for specified *d*

The LET (Logical Execution Time) Assumption



Contrast LET with the Standard Practice



LET-based Real-Time Programming

The programmer specifies *d* to solve the control problem at hand.

The compiler ensures that *d* is met on a given platform; otherwise it rejects the program.

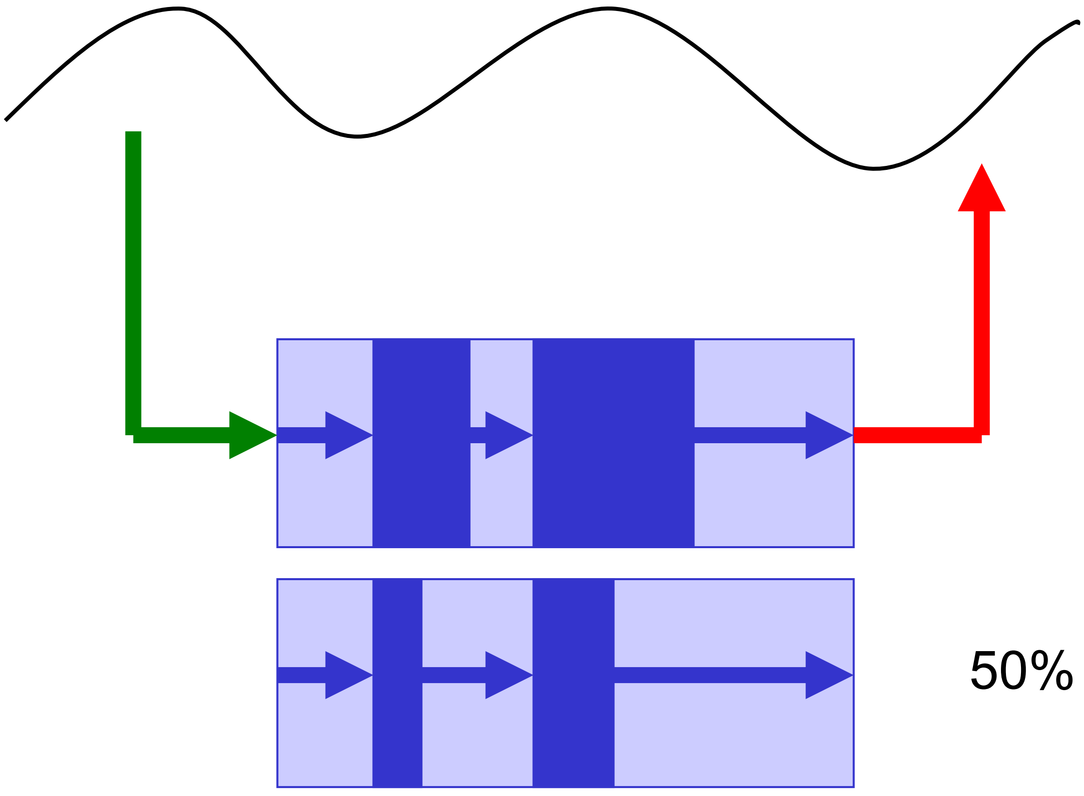
LET-based Real-Time Programming

The programmer specifies *d* to solve the control problem at hand.

The compiler ensures that *d* is met on a given platform; otherwise it rejects the program.

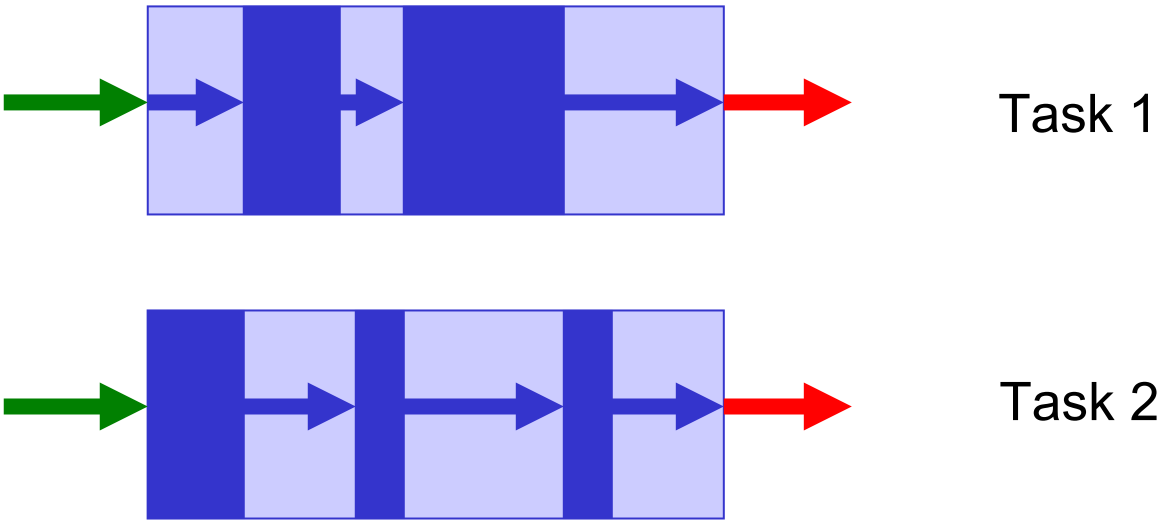
The proof that *d* is met (proof of "time safety") produces a schedule $\rightarrow$ Schedule Carrying Code [H, Kirsch, Matic].

Portability

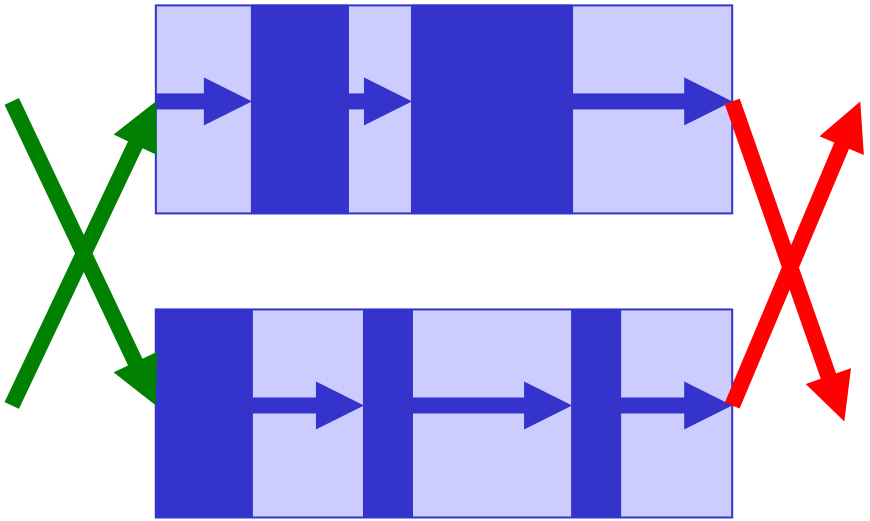


50% CPU speedup

Composability



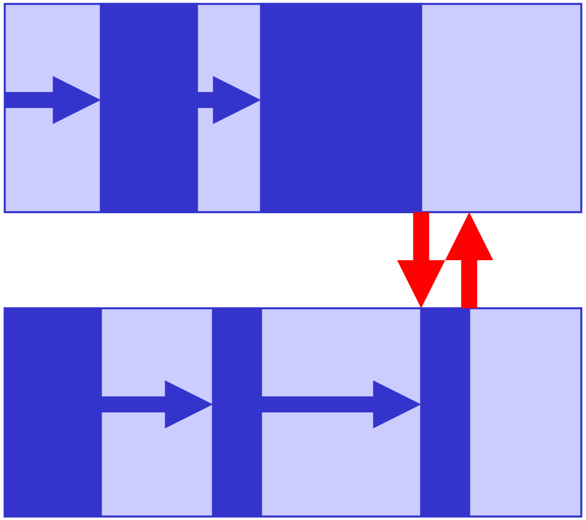
Predictability / Verifiability



-timing predictability: minimal jitter
-value predictability: no race conditions

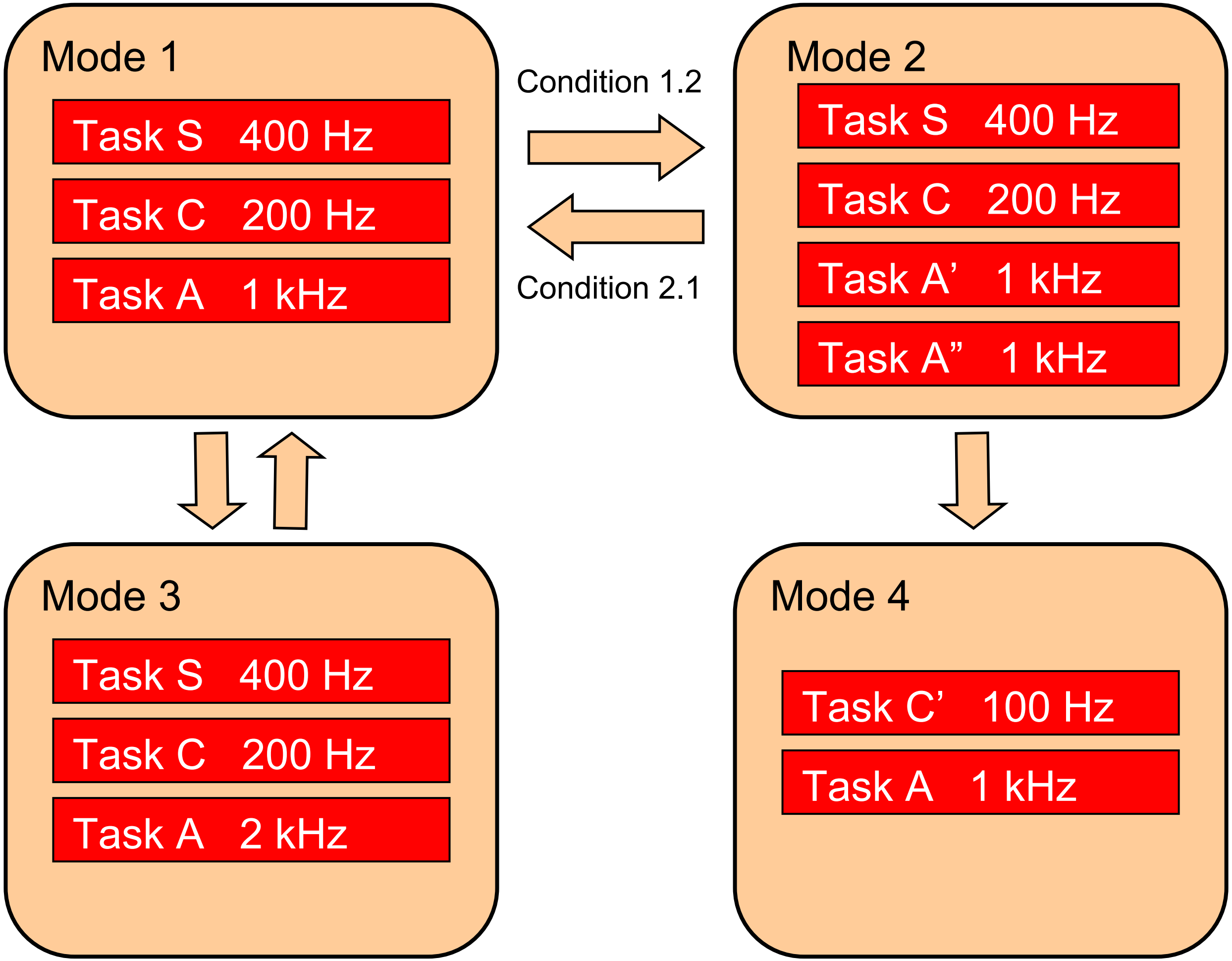
Environment determined behavior!

Contrast LET with the Standard Practice



Race

GIOTTO:
LET for periodic tasks with time-triggered mode switching



xGIOTTO:

Event-Triggered LET Programming [Ghosal, H, Kirsch, Sanvido]

If all events can happen at any time, then few programs would be time-safe.

However, nested reaction blocks can specify the selective listening to events ("event scoping") $\rightarrow$ Structured LET Programming.

xGIOTTO:

Event-Triggered LET Programming [Ghosal, H, Kirsch, Sanvido]

1. Schedule Instruction:

   schedule Task **by** Event **;**

   logical deadline

2. Reaction Block:

   react {
   when Event **do** Block **;**
   whenever Event **do** Block **;**
   begin … end ;
   } until Event **;**

Our Research Explores Three Paradigms

In modeling, use discounted quantitative measures.

In composition, treat inputs and outputs contra-variantly.

In implementation, preserve logical execution times.

In composition, treat inputs and outputs contra-variantly.

This seems obvious:

The "type" of a component should be

$$\text{Inputs} \to \text{Outputs}$$

not

$$\text{Inputs} \times \text{Outputs}.$$

(These two are the same in set theory, but not in type theory!)

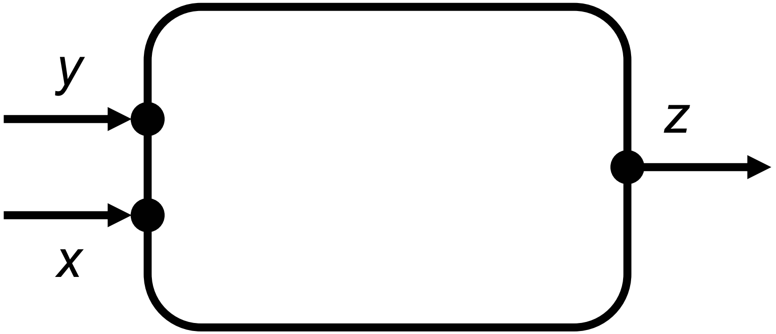
In composition, treat inputs and outputs contra-variantly.

Surprisingly, this is rather non-standard:

If your notion of composition is intersection or product,

or your notion of refinement / abstraction is simulation or language containment,

then you treat inputs and outputs co-variantly (and are in good company)!



Input constraint:
not $x=y=1$

Output constraint:
none

This is an assumption about the environment.

This is an abstraction of the component.

Input constraint:
not x=y=1

Possible behaviors:

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1

Output constraint:
none

Compose with *y*=*z*, forgetting what is input, what output.

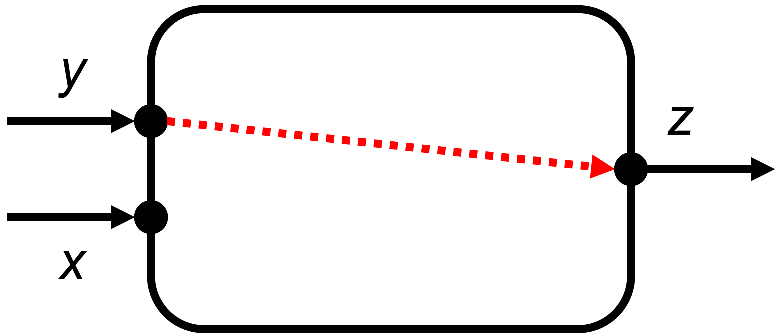
Input constraint:
not *x*=*y*=1

Possible behaviors:

x	*y*	*z*
0	0	0
~~0~~	~~0~~	~~1~~
~~0~~	~~1~~	~~0~~
0	1	1
1	0	0
~~1~~	~~0~~	~~1~~

Output constraint:
none

Compose with y=z, constraining only output (the component).



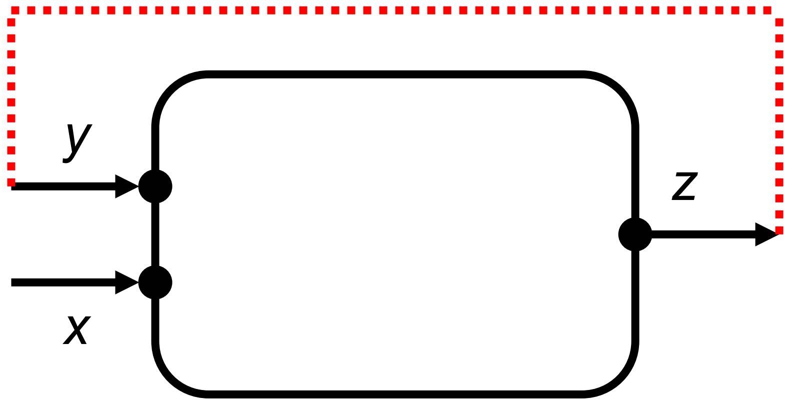
Input constraint:
not x=y=1

Possible behaviors:

x	y	z
0	0	0
~~0~~	~~0~~	~~1~~
~~0~~	~~1~~	~~0~~
0	1	1
1	0	0
~~1~~	~~0~~	~~1~~

New output constraint:
z=y

Compose with *y*=*z*, constraining only inputs (the environment).



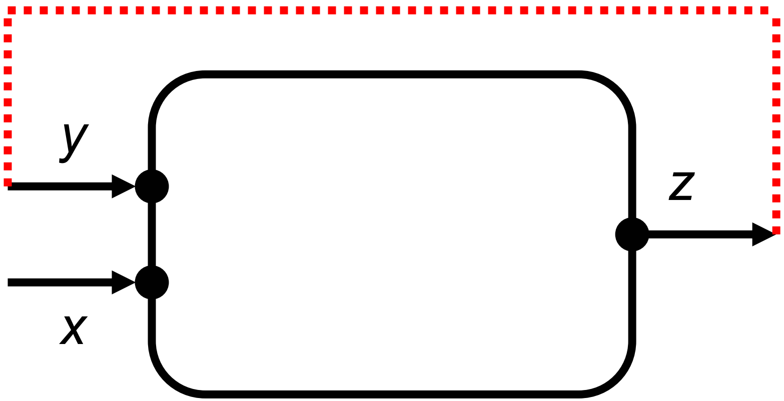
Input constraint:
not x=y=1

Possible behaviors:

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1

Output constraint:
none

Compose with $y=z$, constraining only inputs (the environment).



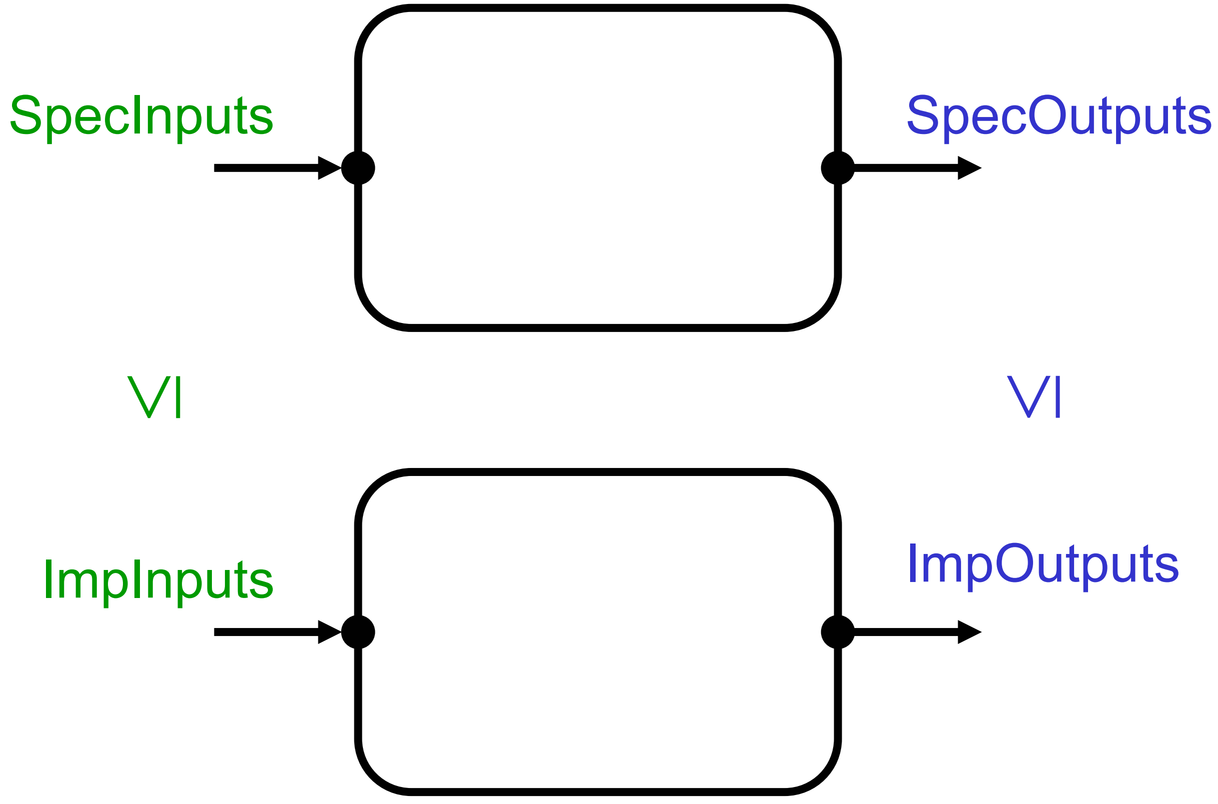
New input constraint:
$x=0$

Possible behaviors:

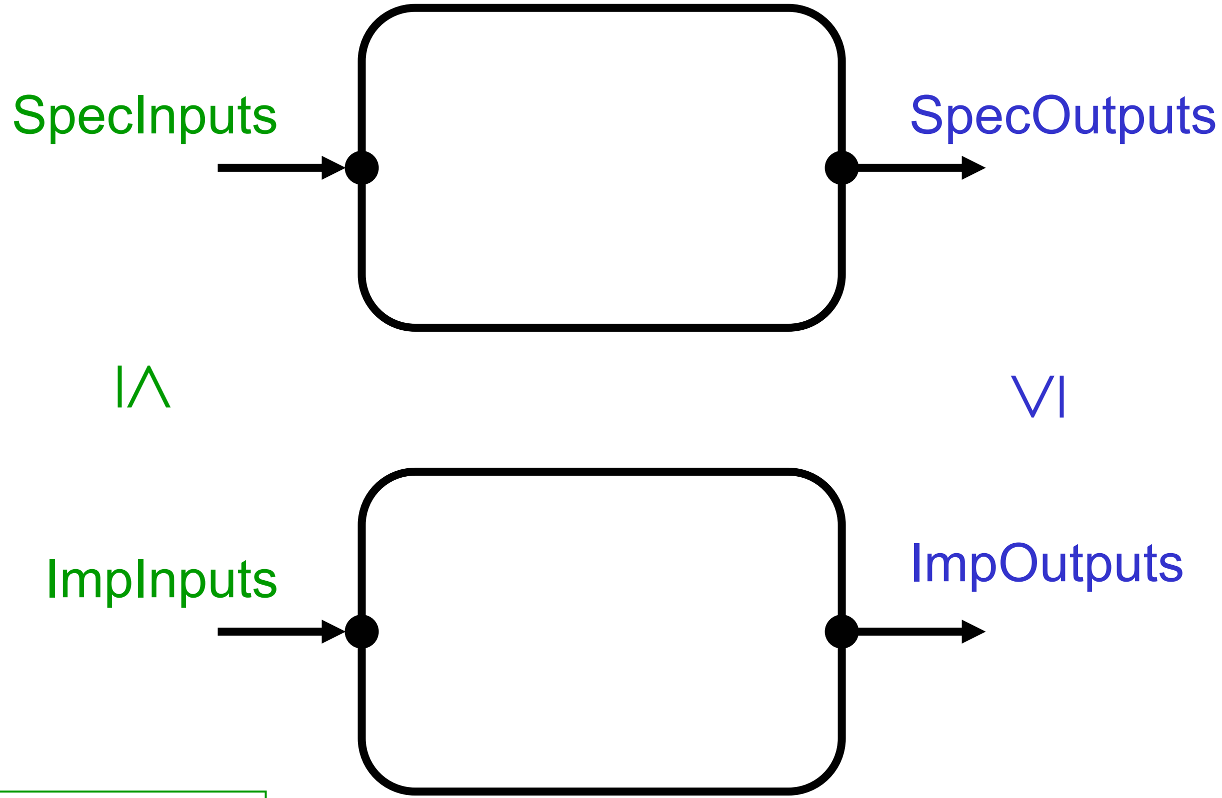
x	y	z
0	0	0
~~0~~	~~0~~	~~1~~
~~0~~	~~1~~	~~0~~
0	1	1
~~1~~	~~0~~	~~0~~
~~1~~	~~0~~	~~1~~

Output constraint:
none

Traditional Behavioral Refinement: Simulation or Language Containment

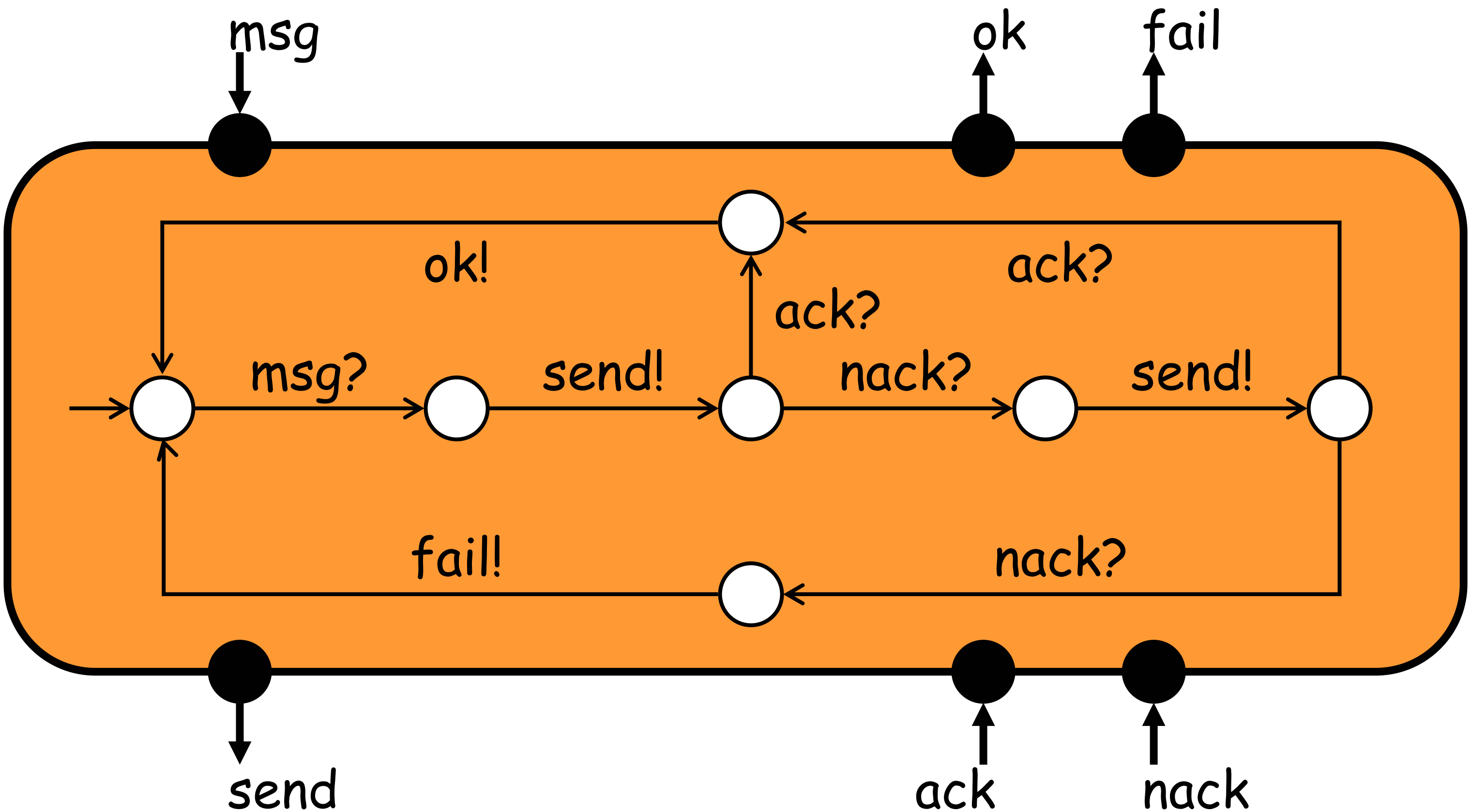


Contra-variant Refinement:
Implementations can be substituted for Specifications

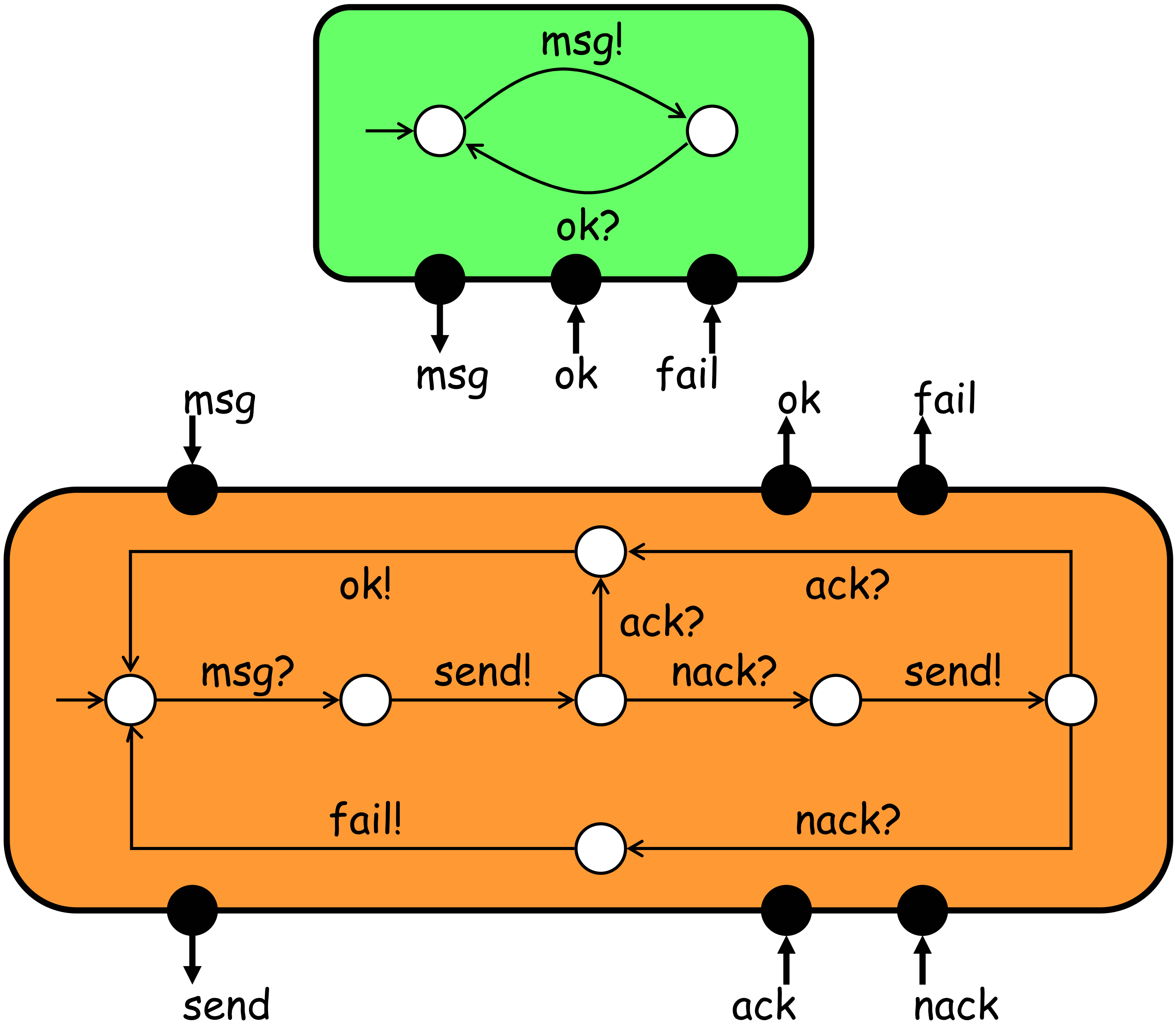


Implementation accepts all legal inputs.

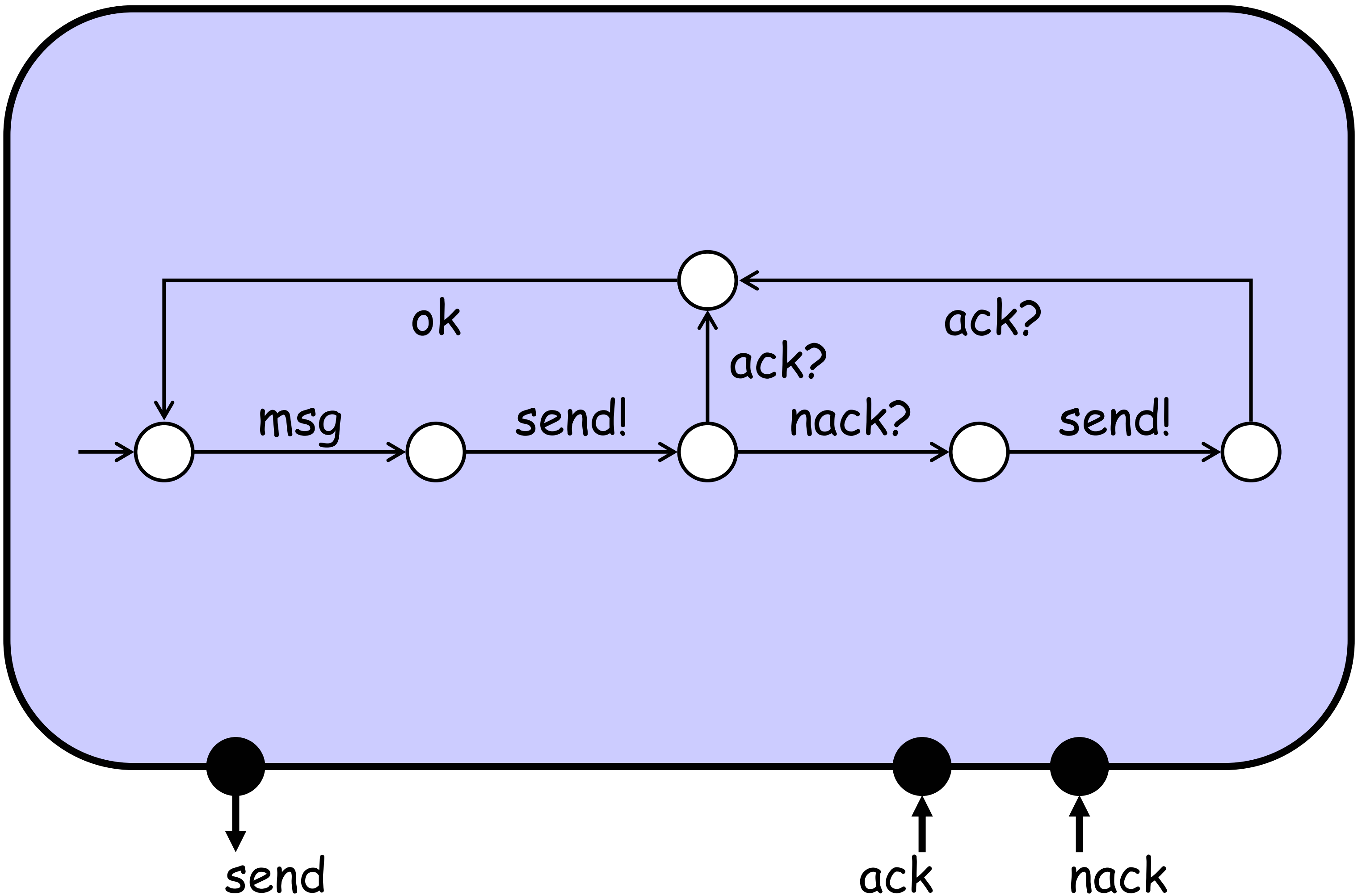
Implementation produces only legal outputs.

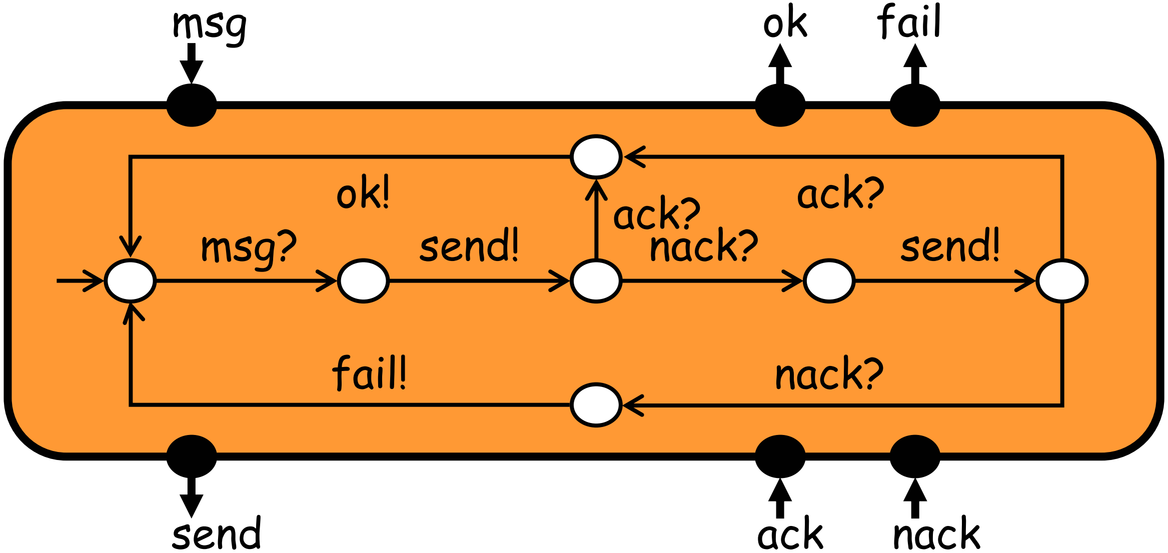


msg
ok
fail
ok!
ack?
ack?
msg?
send!
nack?
send!
fail!
nack?
send
ack
nack


msg!
ok?
msg
ok
fail
msg
ok
fail
ok!
ack?
ack?
msg?
send!
nack?
send!
fail!
nack?
send
ack
nack

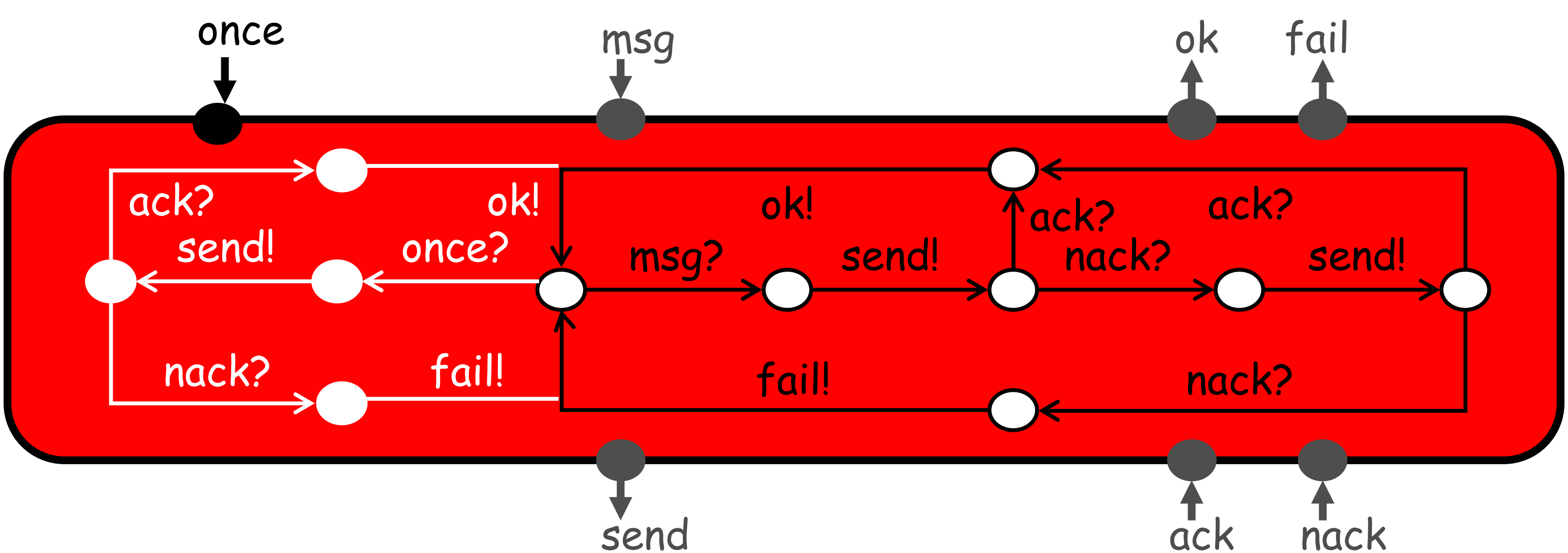
The Composite Interface





msg
ok
fail
ok!
ack?
ack?
msg?
send!
nack?
send!
fail!
nack?
send
ack
nack

VI



once
msg
ok
fail
ack?
ok!
ok!
ack?
ack?
send!
once?
msg?
send!
nack?
send!
nack?
fail!
fail!
nack?
send
ack
nack

We call a formalism with

-input-constraining composition
-contra-variant refinement

an interface theory [de Alfaro, H].

We call a formalism with

-input-constraining composition
-contra-variant refinement

an interface theory [de Alfaro, H].

We have developed several interface theories, e.g. for

-message-passing components ("interface automata")

-synchronous hardware components [Chakrabarti, dA, H, Mang]

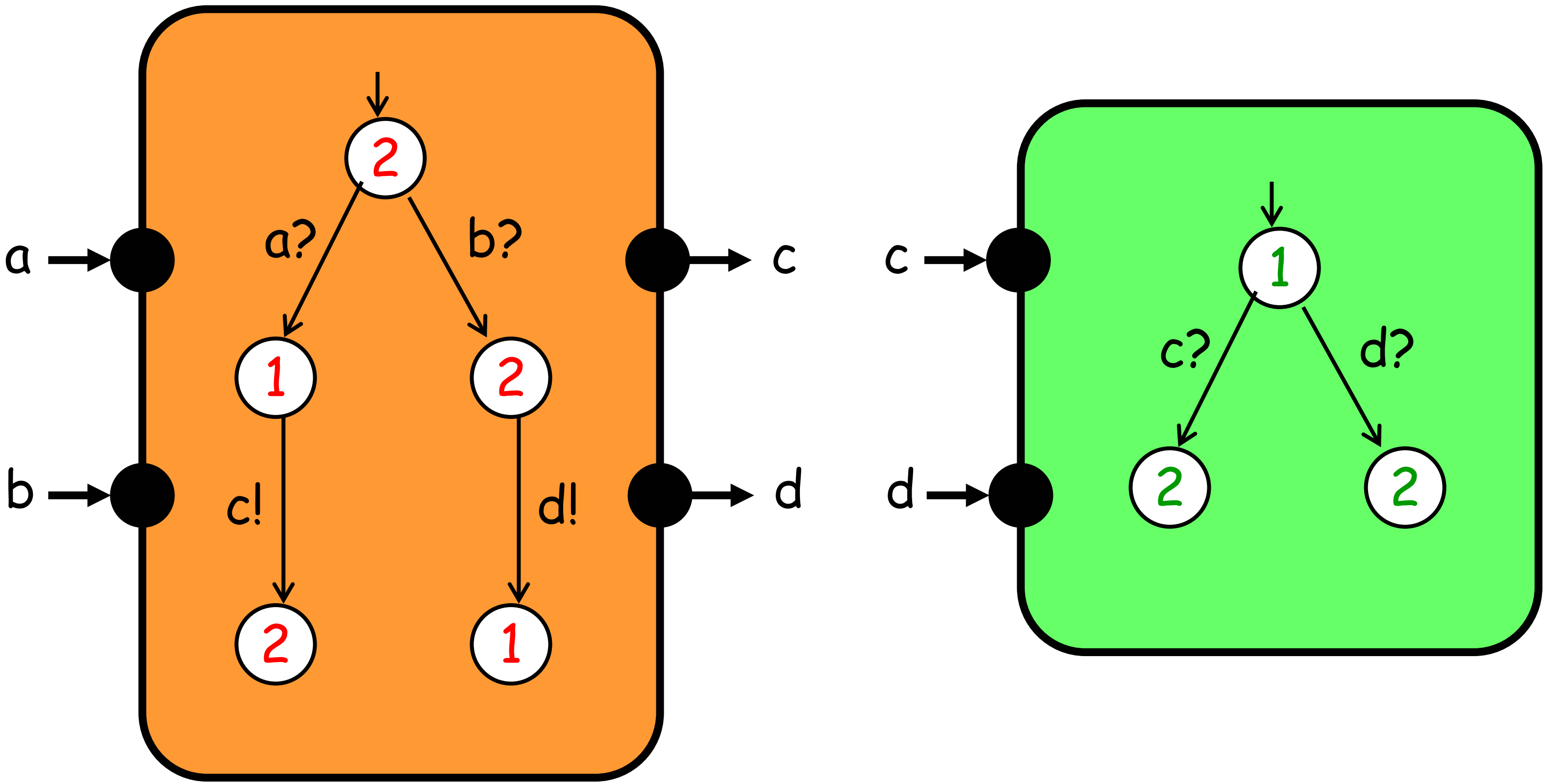
-possibly recursive software modules [C, dA, H, Jurdzinski, M]

-real-time components [dA, H, Stoelinga]

-resource-constrained components [C, dA, H, S]

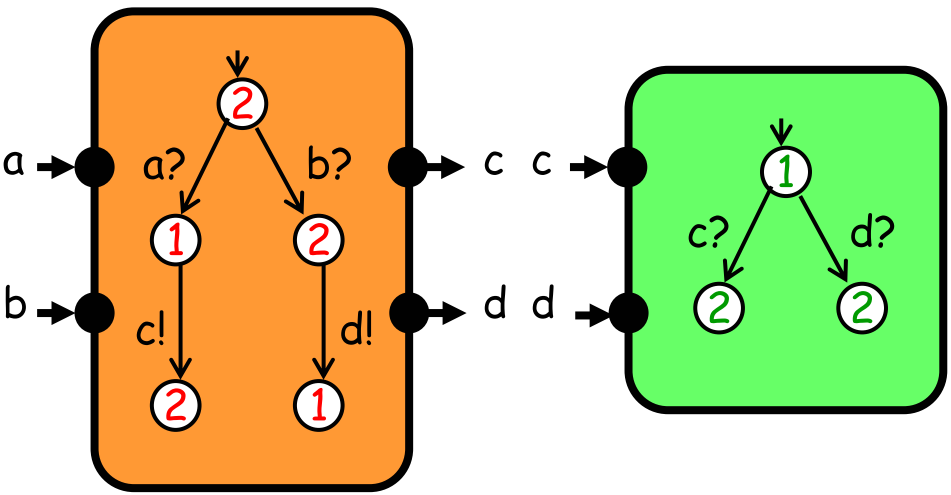
They have been implemented in the CHIC tool.

Resource Interfaces

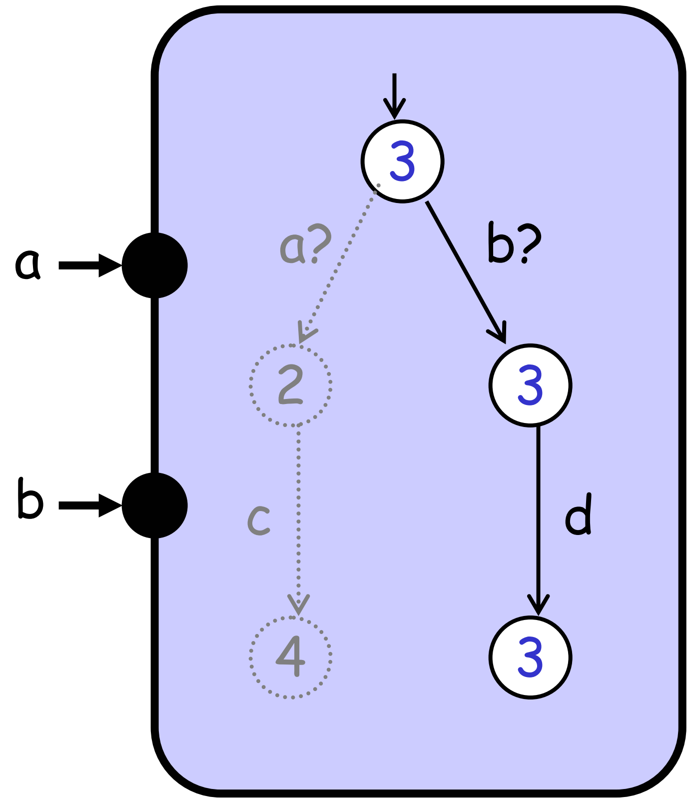


Available peak power: 3

Resource Interfaces



Available peak power: 3



The composite interface.

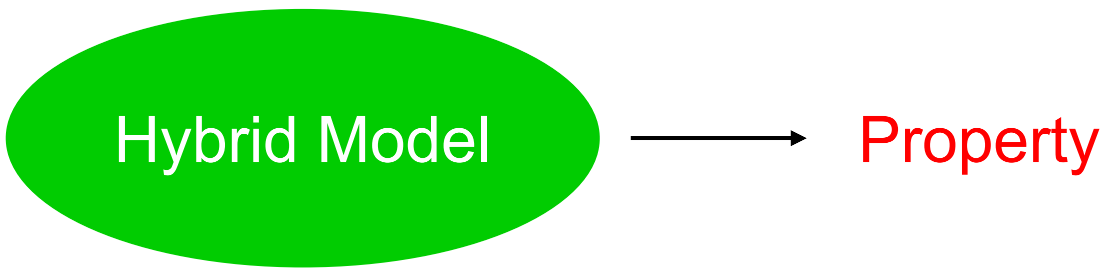
Our Research Explores Three Paradigms

In modeling, use discounted quantitative measures.

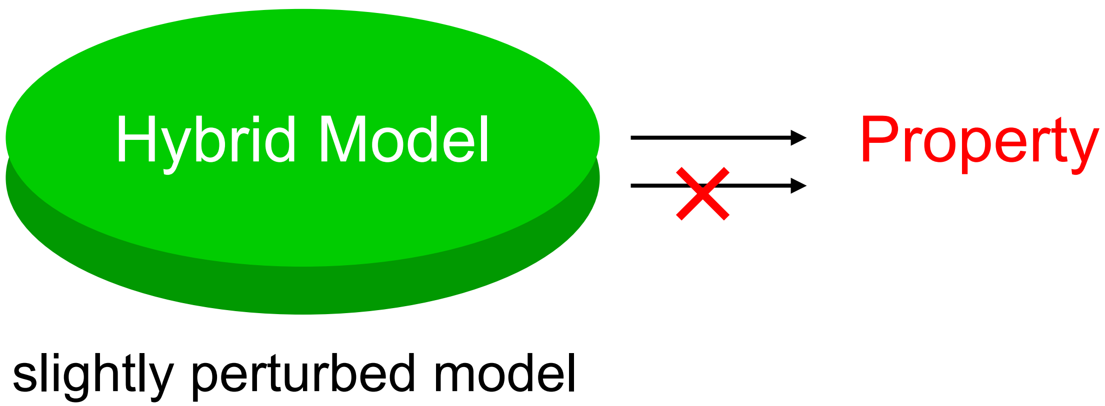
In composition, treat inputs and outputs contra-variantly.

In implementation, preserve logical execution times.

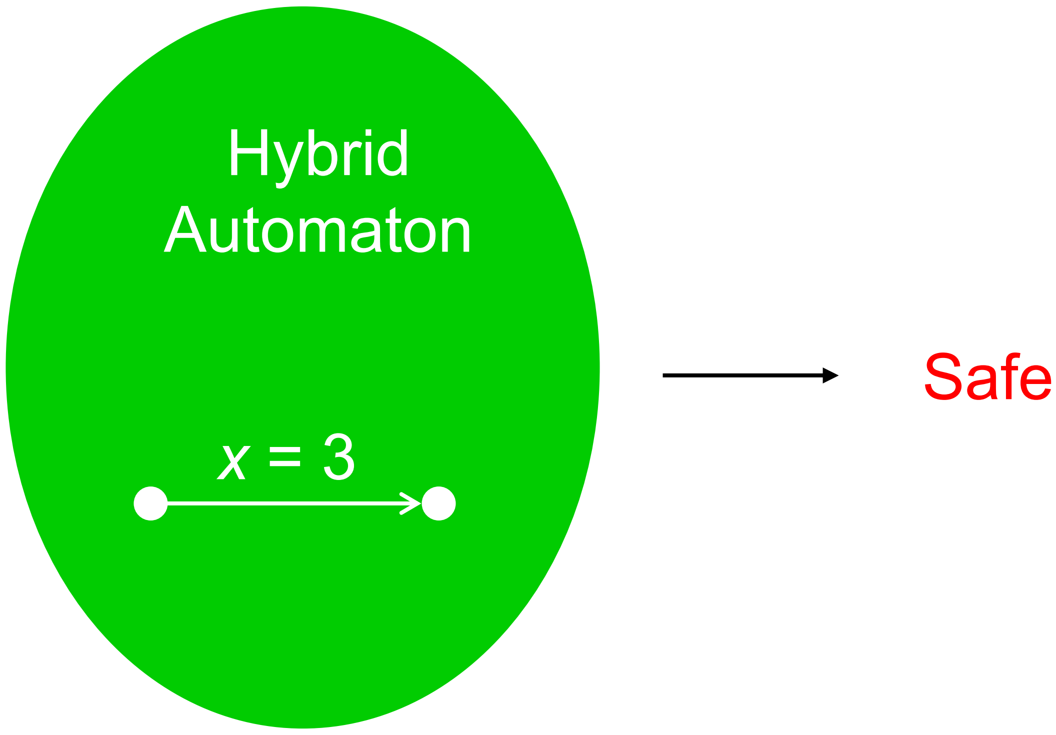
The Problem



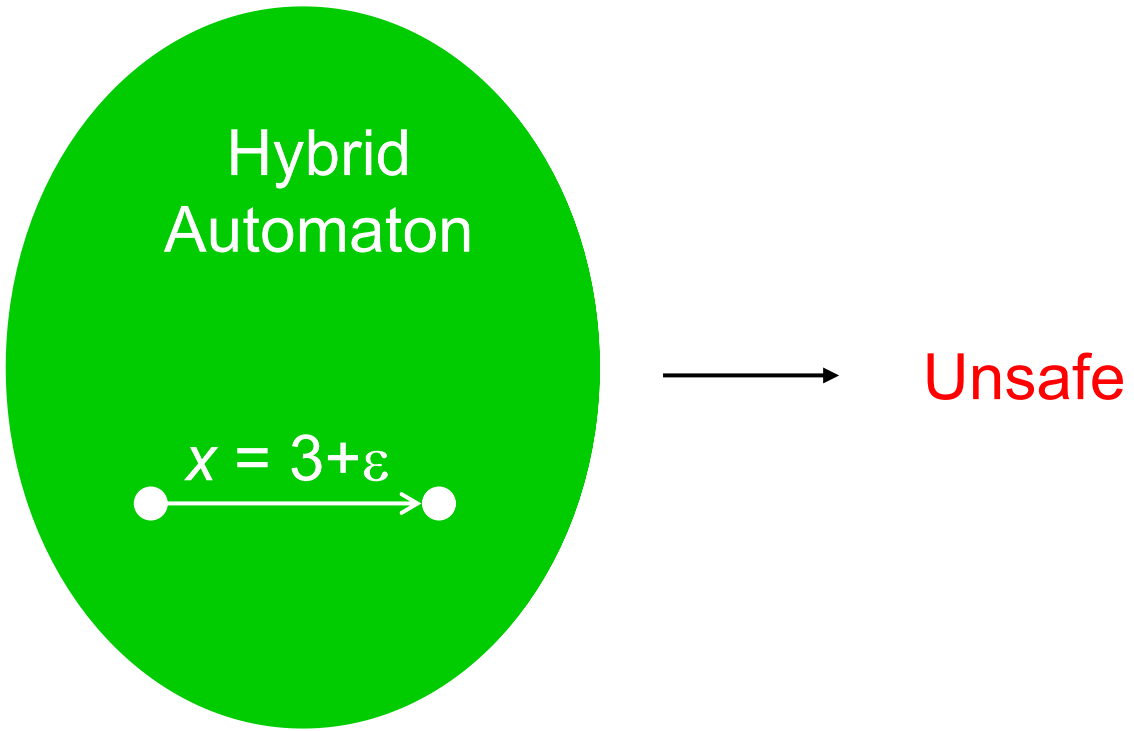
The Problem



The Problem



The Problem



The Proposed Solution

value(Model,Property): States $\rightarrow$ {0,1}

value(Model,Property): States $\rightarrow$ [0.1]

Discounting the Future

value(Model,Property): States $\rightarrow$ {0,1}

$\text{value}(m, \Diamond T) = \mu X.\ (T \vee \text{pre}(X))$

discountedValue(Model,Property): States $\rightarrow$ [0,1]

$\text{discountedValue}(m, \Diamond T) = \mu X.\ \max(T,\ \lambda \cdot \text{pre}(X))$

discount factor $0<\lambda<1$

Robustness Theorem:

If discountedBisimilarity(m_1,m_2) > 1 - ε,
then |discountedValue(m_1,p) - discountedValue(m_2,p)| < $f(\varepsilon)$.

Robustness Theorem:

If discountedBisimilarity(m_1,m_2) > 1 - ε,
then |discountedValue(m_1,p) - discountedValue(m_2,p)| < $f(\varepsilon)$.

Further Advantages of Discounting:

-approximability because of geometric convergence (avoids non-termination of fixpoint iteration)

-applies also to probabilistic systems and to games (enables control)

Our Research Explores Three Paradigms

In modeling, use discounted quantitative measures.

ICALP '03 [de Alfaro, H, Majumdar]

In composition, treat inputs and outputs contra-variantly.

www.eecs.berkeley.edu/~tah/chic

In implementation, preserve logical execution times.

www.eecs.berkeley/~tah/giotto