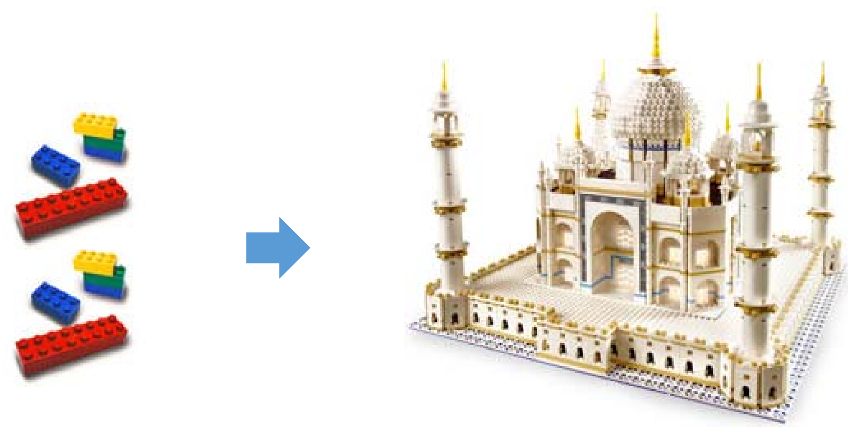
The Science of Software and System Design

Stavros Tripakis

Aalto University and UC Berkeley



WODES plenary, May 30, 2018

Bridging the gap between DES and formal methods

Discrete Event Dynamic Systems
June 2017, Volume 27, Issue 2, pp 209–260 | Cite as

Supervisory control and reactive synthesis: a comparative introduction

Authors   Authors and affiliations

Rüdiger Ehlers, Stéphane Lafortune, Stavros Tripakis, Moshe Y. Vardi

Article
First Online: 15 March 2016

606 Downloads   4 Citations

Abstract

This paper presents an introduction to and a formal connection between synthesis problems for discrete event systems that have been considered, largely separately, in the two research communities of *supervisory control* in control engineering and *reactive synthesis* in computer

Cyber-physical systems: future

Courtesy `https://vimeo.com/bsfilms`
Thanks to Christos Cassandras for recommending this video

Cyber-physical systems: present

Autonomous car driving through red light

How do we typically design systems?

- The *trial and error* approach:
  - Build → test → fix → repeat.
- Problems with this approach:
  - Un-scalable
  - Un-economic
  - Un-safe
- Yet common…

Are we the drivers supposed to debug the autopilot?

*How to design **better systems**?*
*How to **better design** systems?*

"It was described as a beta release. The system will learn over time and get better and that's exactly what it's doing. It will start to feel quite refined within a couple of months." – Elon Musk, Tesla CEO, Nov 2015

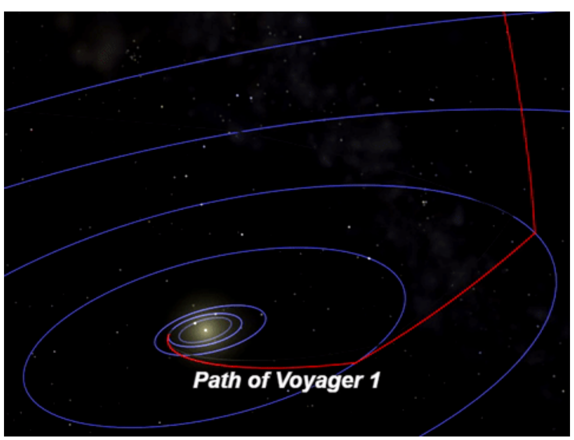
Tesla driver dies in first fatal crash while using autopilot mode

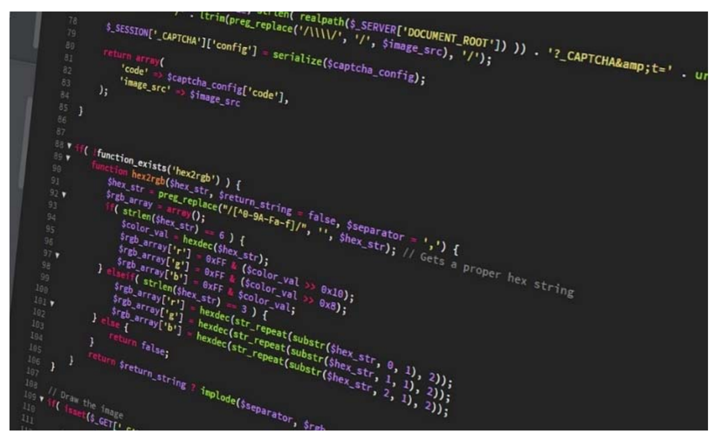
June 2016

The autopilot sensors on the Model S failed to distinguish a white tractor-trailer crossing the highway against a bright sky

Is system design an art or a science?

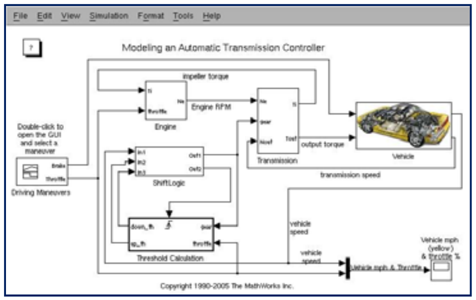
- Science = knowledge that helps us make predictions
- We can make fairly good predictions about several systems we build (buildings, bridges, satellites, …)
- What can we say about cyber-physical systems?
- What predictions can we make about software? (term used broadly)





Elements of the science of system design (“model-based design”)

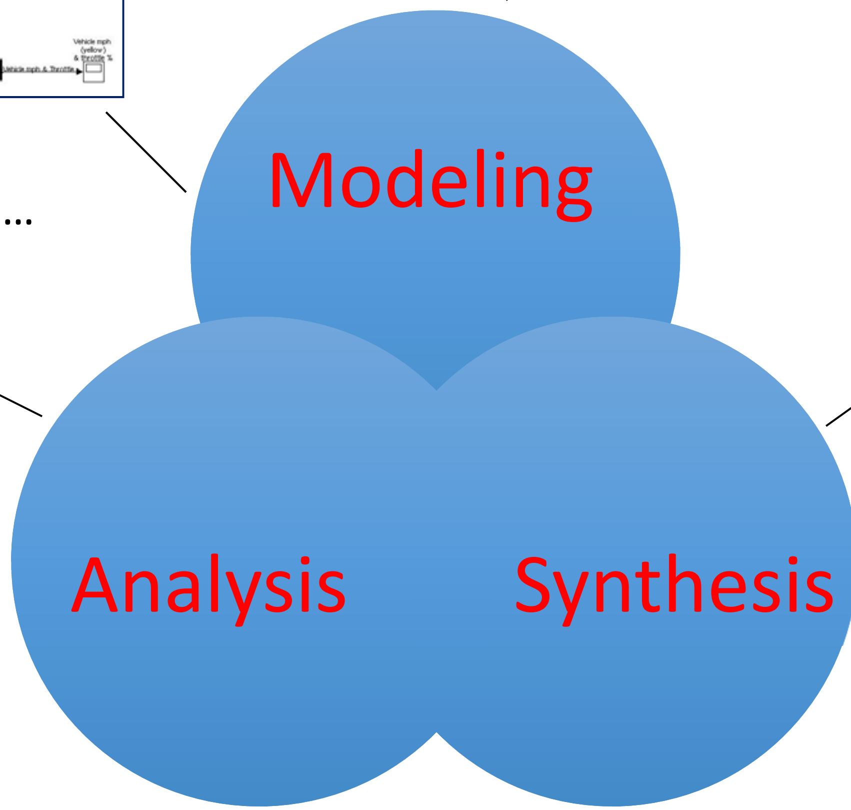
Simulink, UML, SysML, HDLs, SystemC, …

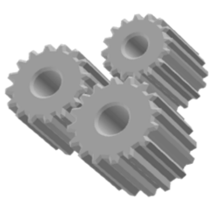


Describe the system that we want

Simulation, verification, …

Be sure that this is what we want





Implement the system
Automatically
Correct-by-construction

Tripakis

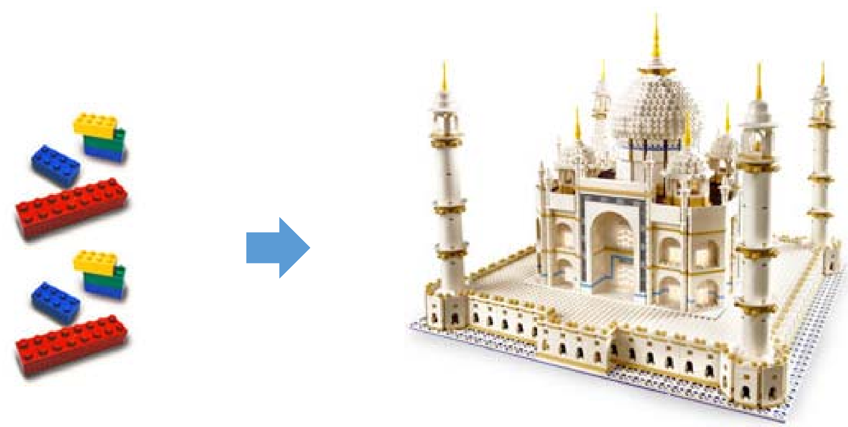
This talk: some recent work

- The Refinement Calculus of Reactive Systems
- Synthesis of platform mappings with applications to security
- (Time permitting) Combining controller synthesis and learning
  - or *Why model-based design is not the end of the story*

The Refinement Calculus of Reactive Systems (RCRS)

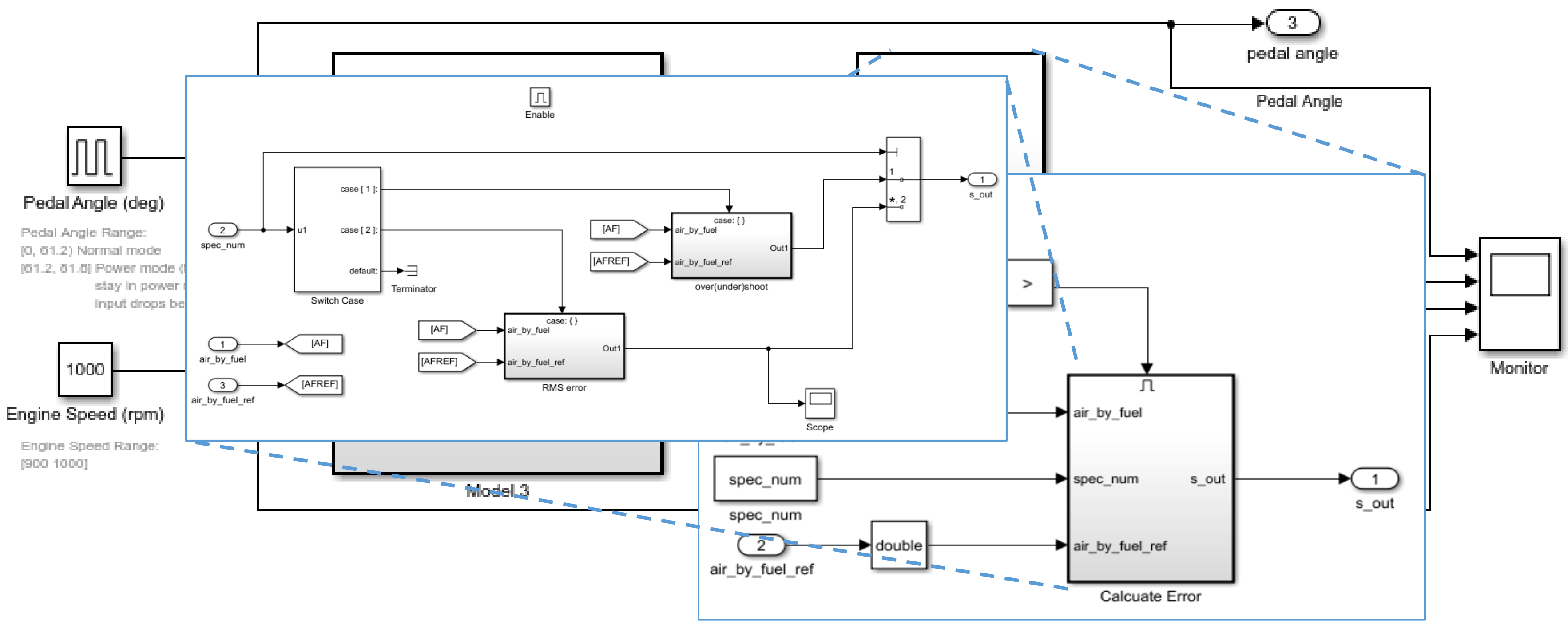
Joint work with Viorel Preoteasa and Iulia Dragomir (Aalto)

Sponsors: Academy of Finland and NSF CPS Breakthrough



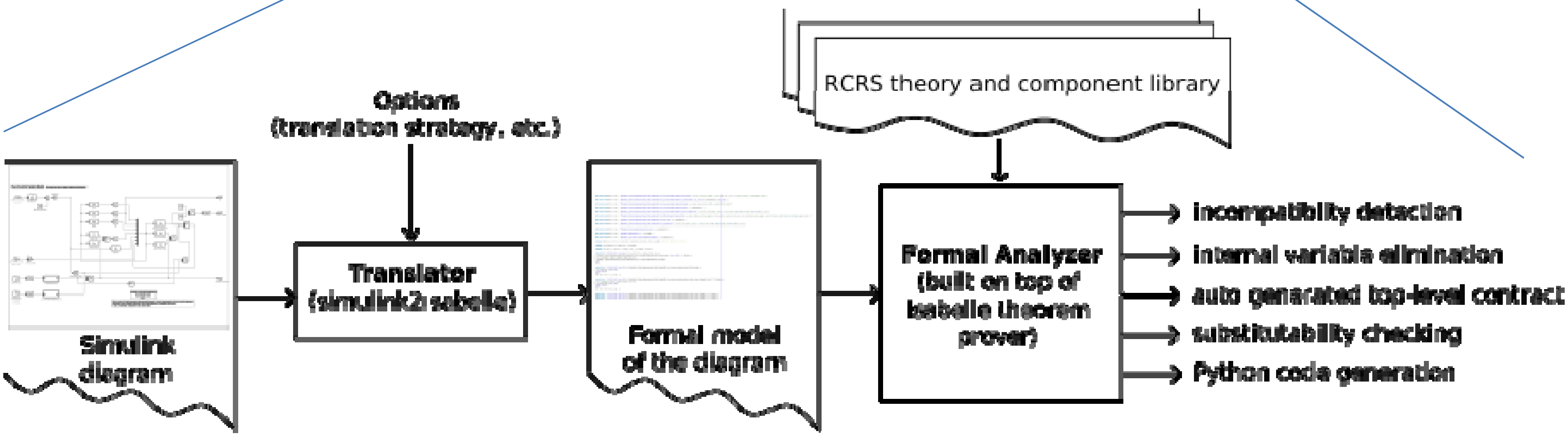
Motivation

- Compositional formal reasoning for CPS – Simulink:



Distinguished Artifact Award
TACAS 2018

RCRS = theory + toolset



Downloadable from http://rcrs.cs.aalto.fi/

RCRS theory: contract-based design

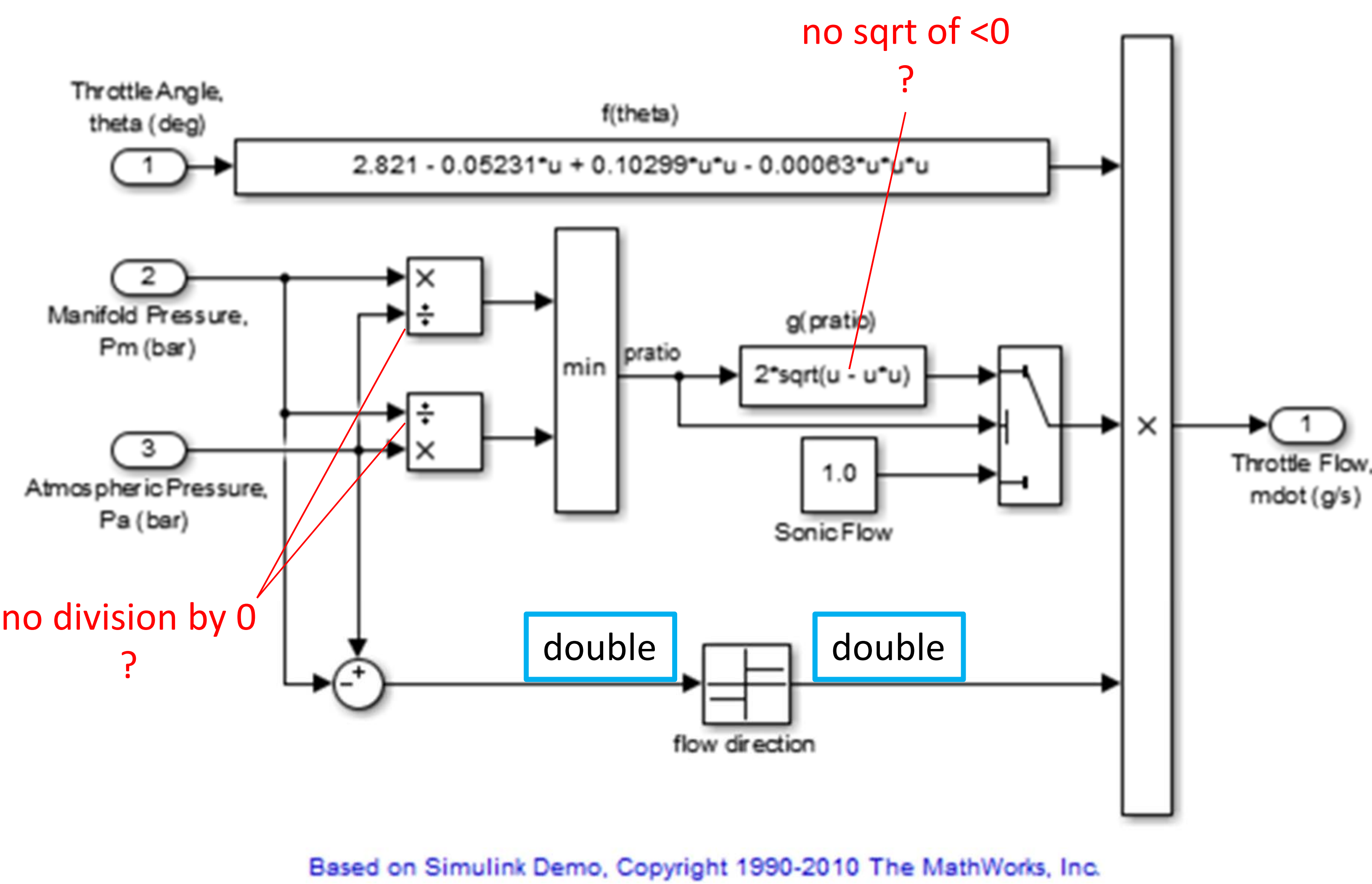
- Relational interfaces [EMSOFT'09, ACM TOPLAS'11]
  - Symbolic, synchronous version of interface automata [Alfaro, Henzinger]
  - Open, non-deterministic, non-input-complete systems
    (this is crucial for static analysis)
  - Semantic foundation: relations
  - Limited to safety properties

- Refinement calculus of reactive systems [EMSOFT'14]
  - Richer semantics: predicate and property transformers
  - Can handle both safety and liveness properties
  - Entirely formalized in Isabelle theorem prover - 27k lines of Isabelle code

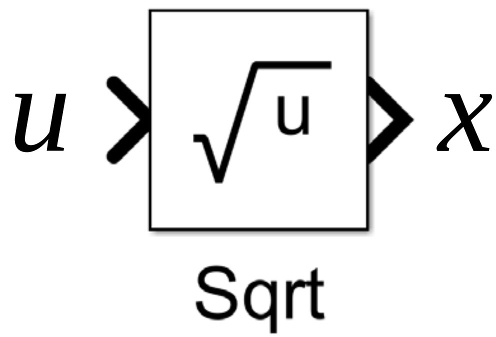
Some of the things RCRS can do

- Design-by-contract
- Incremental design with refinement
- Compositional verification

Static ("compile-time") analysis



Simulink square root modeled with RCRS contracts



`double -> double`

Simulink type

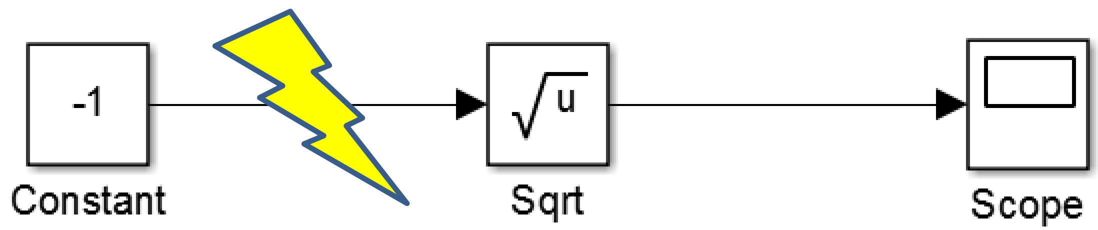
$$u \geq 0 \rightarrow x = \sqrt{u}$$

RCRS contract:
input-receptive

$$u \geq 0 \wedge x = \sqrt{u}$$

RCRS contract:
non-input-receptive

Catching incompatibilities statically

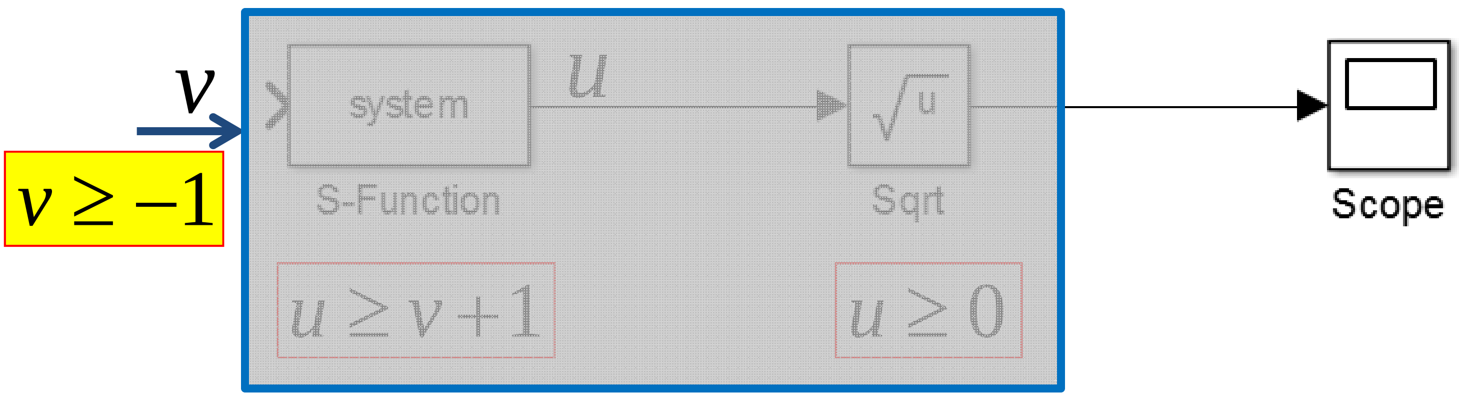


$u = -1$

$u \geq 0 \wedge x = \sqrt{u}$

caught by taking the conjunction of the two formulas
and checking satisfiability

Inferring new contracts automatically

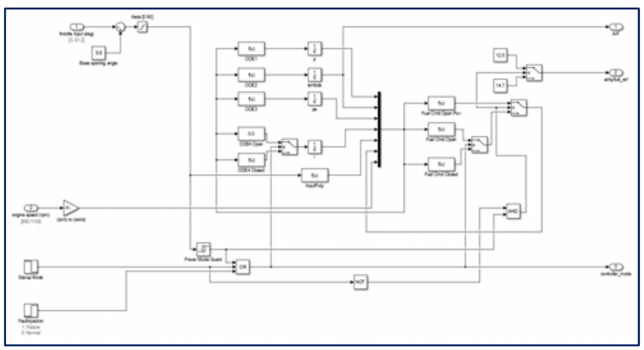


Library of Simulink basic blocks in RCRS

```
definition "Id = [: x ⇝ y . y = x :]"
definition "Add = [: (x, y) ⇝ z . z = x + y :]"
definition "Constant c = [: x::unit ⇝ y . y = c :]"
definition "UnitDelay = [: (x,s) ⇝ (y,s') . y = s ∧ s' = x :]"
definition "Sqrt = {. x . x ≥ 0 .}  o  [: x ⇝ y . y = √x :]"
definition "NonDetSqrt = {. x . x ≥ 0 .}  o  [: x ⇝ y . y ≥ 0 :]"
definition "ReceptiveSqrt = [: x ⇝ y . x ≥ 0 ⟹ y = √x :]"
definition "Integrator dt = [: (x,s) ⇝ (y,s'). y=s ∧ s'=s+x*dt :]"
```

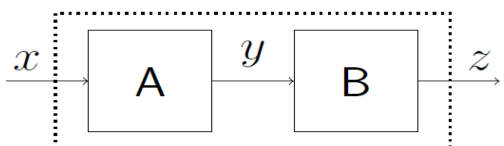
Translation of (arbitrary) Simulink diagrams

- Formal, modular, compositional translation: a non-trivial problem

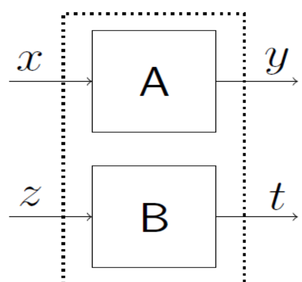


$(\mathsf{Constant} \parallel \mathsf{Constant1}) \circ \mathsf{Div} \circ \mathsf{Scope}$ ...

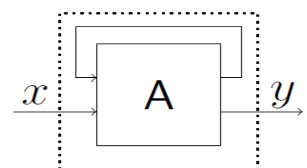
- Algebra of block diagrams:
  - Only 3 composition primitives:



serial



parallel

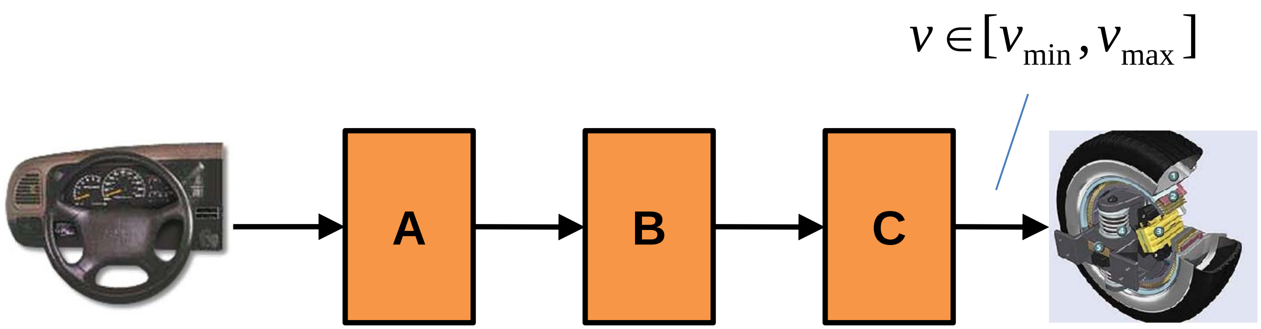


feedback

Defining feedback: non-trivial [LICS'16]

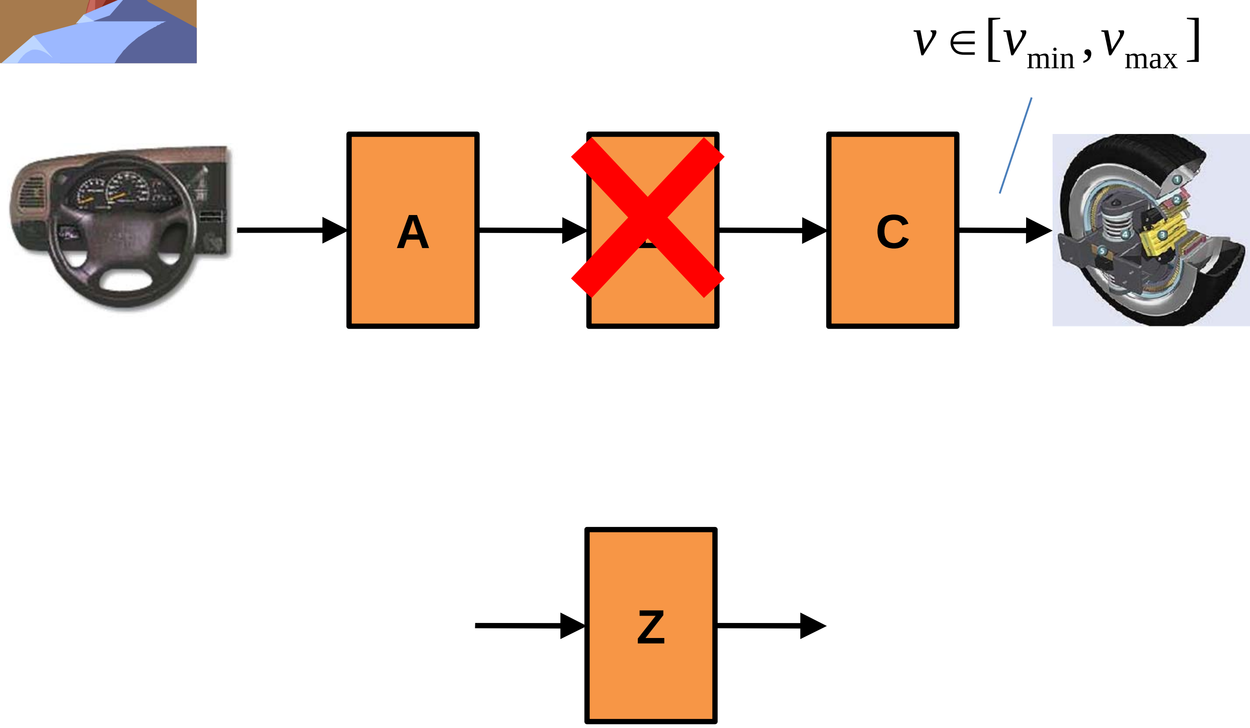
Incremental design with refinement

Suppose we have designed and verified this “steer-by-wire” system:



Incremental design with refinement

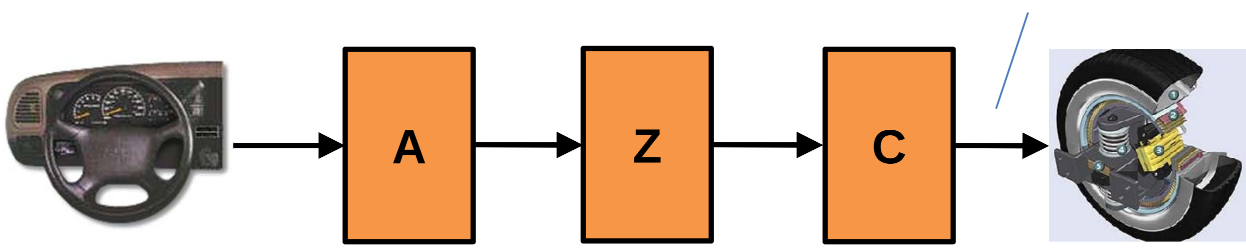
Suppose we want to replace B with Z:



Incremental design with refinement

How to ensure properties are preserved (**substitutability**)?

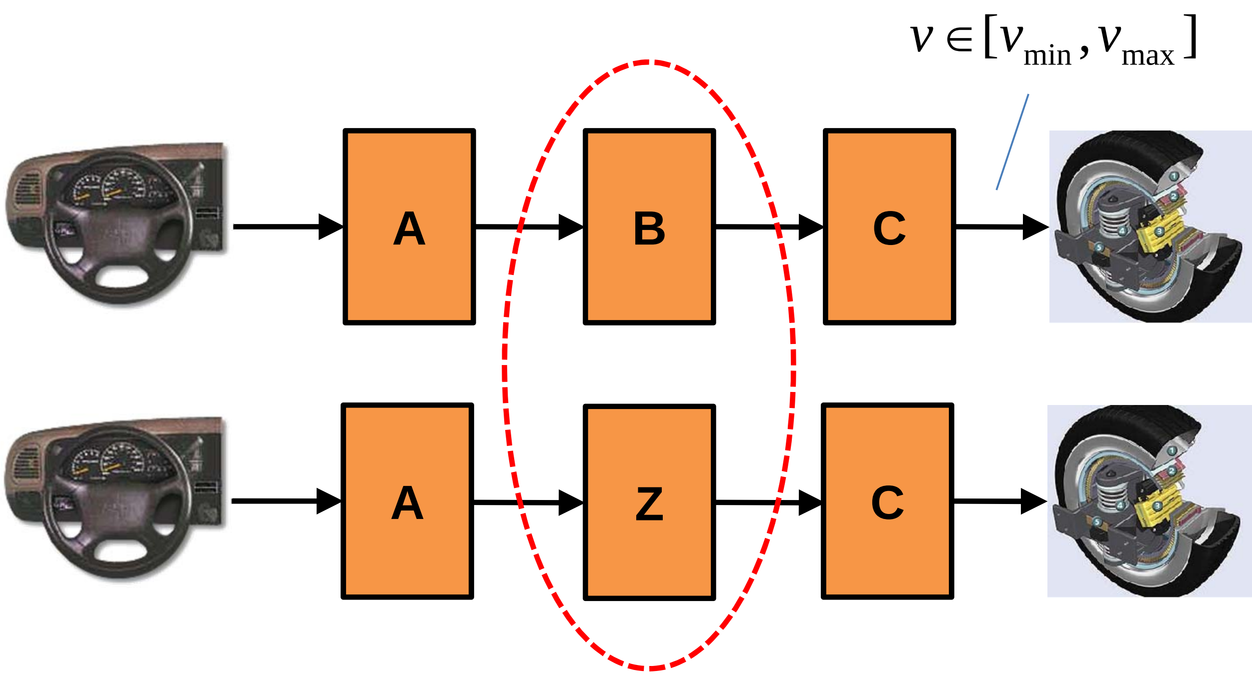
$v \in [v_{min}, v_{max}]$



Incremental design with refinement

In RCRS it suffices to check that

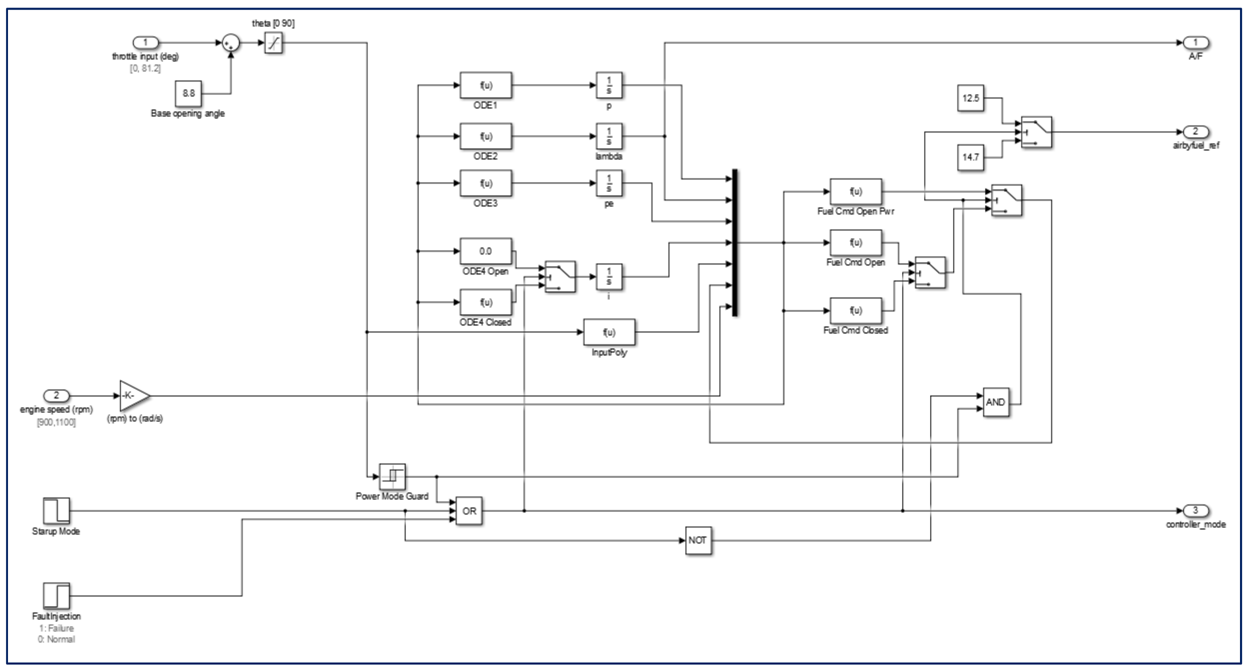
$Z \leq B$: Z *refines* B
(local check)



Does it work for real-world systems?

- Case study: Fuel Control System automotive benchmark
- Made publicly available by Toyota on CPS-VO website
- Simulink model: 3-level hierarchy, 104 blocks
- Translator produces a 1660-line long RCRS theory (translation time negligible)
- Automatic static analysis / contract inference / simplification: <1 minute

Sample subsystem of the FCS model



Synthesis of platform mappings with applications to security

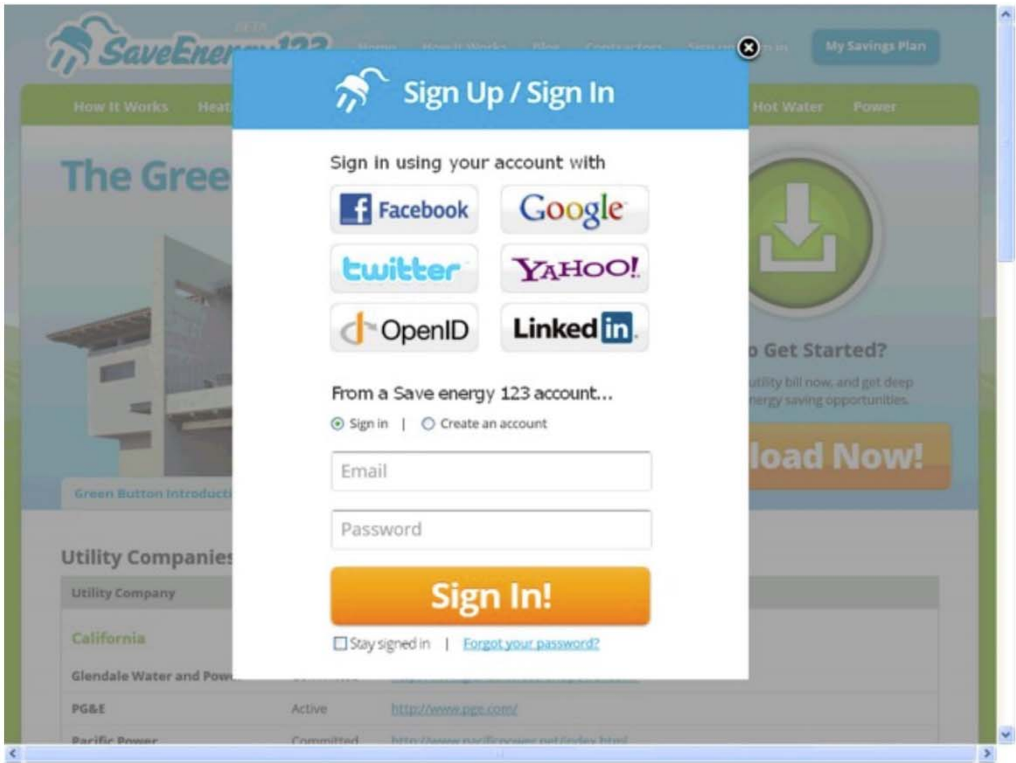
Joint work with Eunsuk Kang (NSF ExCAPE project), and Stephane Lafortune (UMichigan)

Sponsors: NSF Expeditions ExCAPE

Thanks to Eunsuk Kang for several slides

Motivation: security

Third-Party Authentication



OAuth: Widely adapted, support from major vendors
Well-scrutinized & **formally checked**

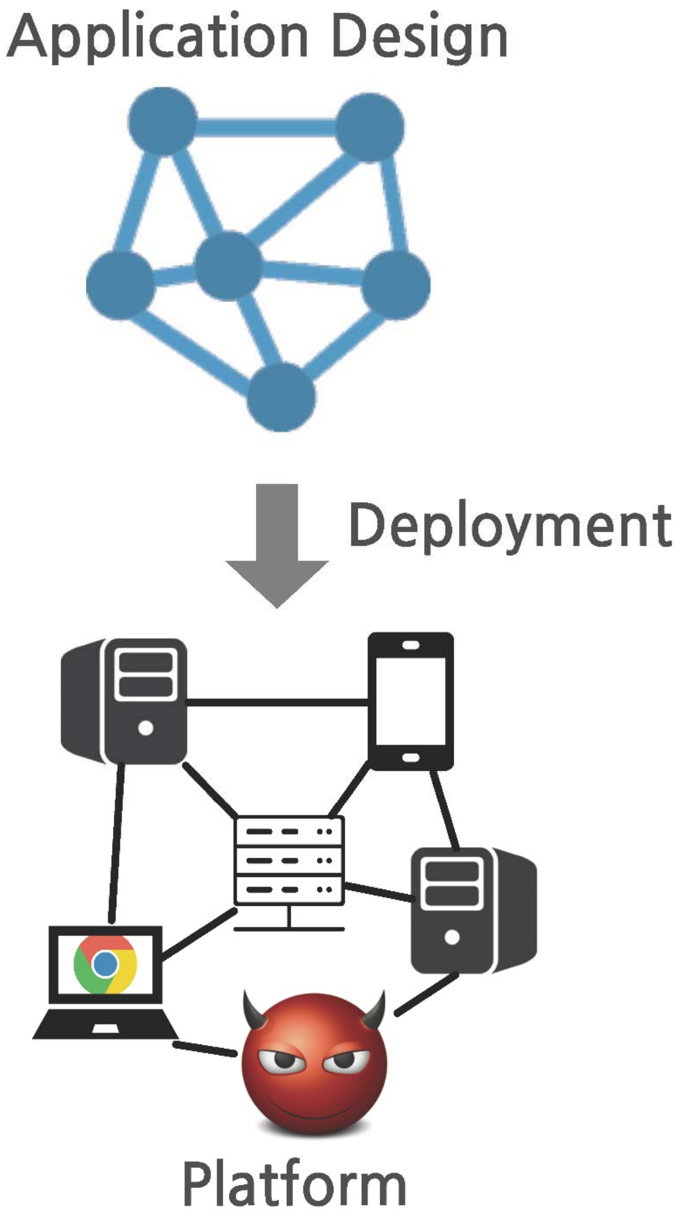
Motivation: security

by **Chris Brook** May 2, 2014 , 1:42 pm

UPDATE — A serious vulnerability in the OAuth and OpenID protocols could lead to complications for those who use the services to log in to websites like Facebook, Google, LinkedIn, Yahoo, and Microsoft among many others.

Study of OAuth providers [Sun & Beznosov, CCS12]
Majority vulnerable (Google, Facebook,...)

The heart of the problem



Designers think at high-level

Protocols, APIs, workflows, use cases, etc.,

Ignore irrelevant details

implementation

Attacks may exploit details absent at high-level

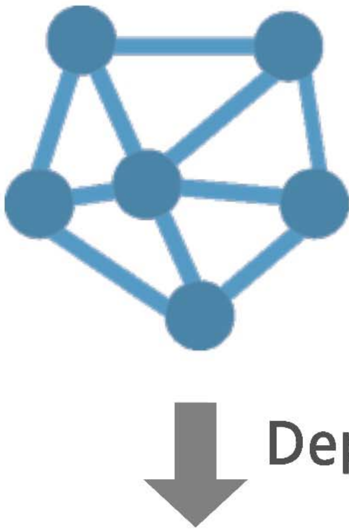
Unwanted features

Unknown environment

Hidden interface/entry points

Our approach: modular modeling with mappings

Application Design



P

Deployment

m

mapping composition operator

$P \;||_m\; Q$

implementation model

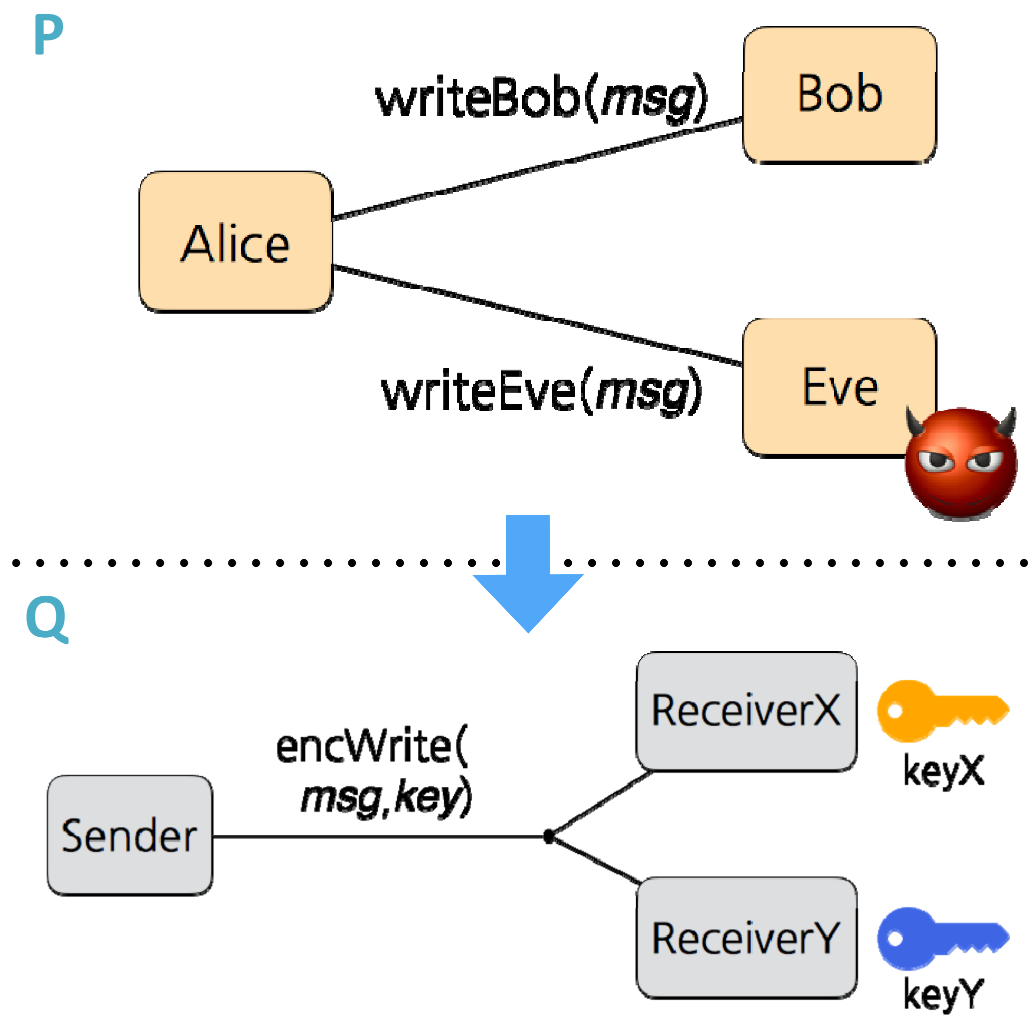
Examples of decisions captured by mappings:

- should a certain protocol message be implemented as an HTTP request?
- with cookies to store secret values?
- with query parameters?

Possible applications beyond security.

Platform

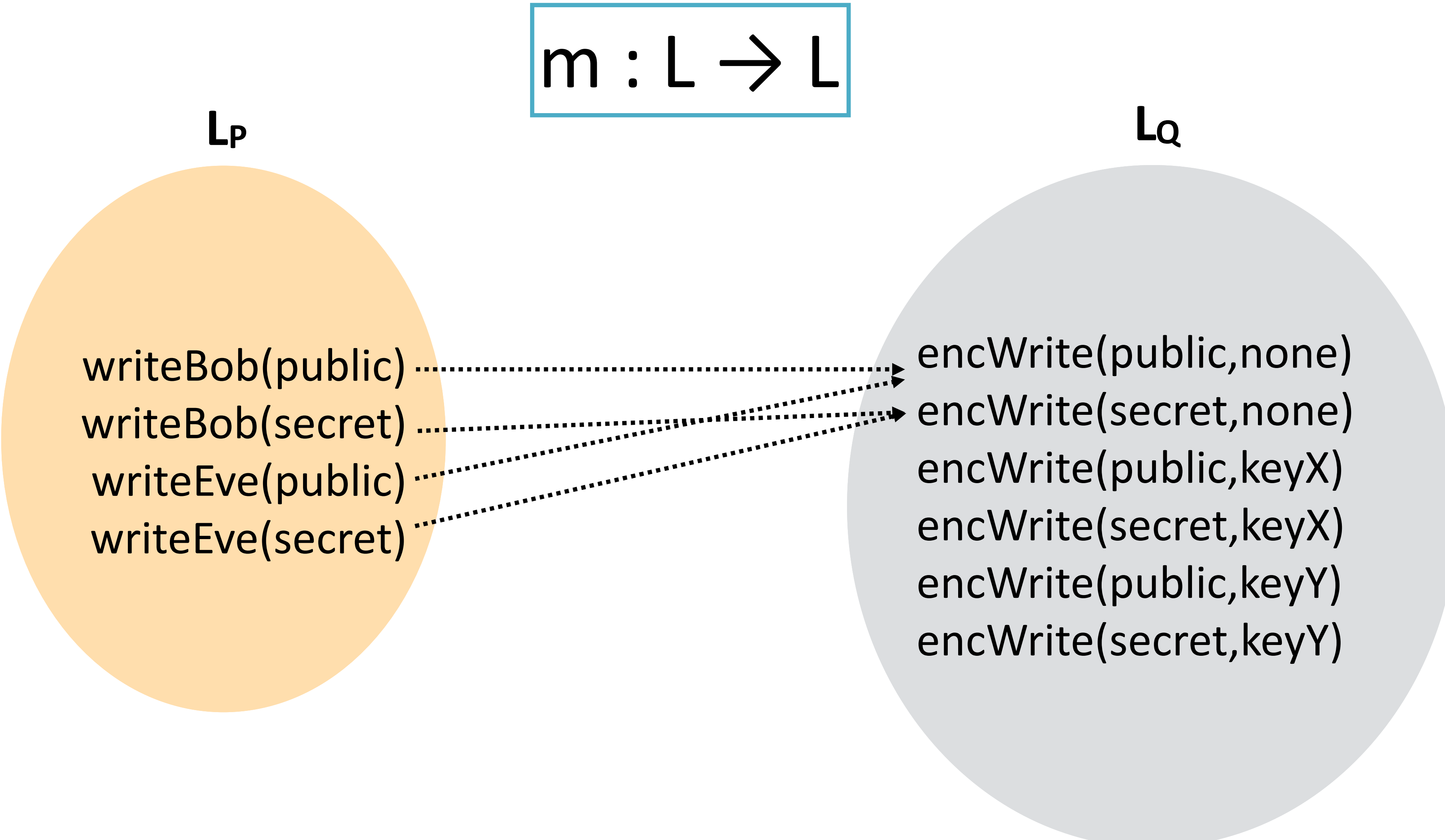
Example: abstract channel & public channel



S ≡ "Only Bob can learn Alice's secret"

Can we implement abstract design **P** on platform **Q** and preserve property **S**?

Implementation decisions as mappings



Correct and incorrect mappings

m_1:

writeBob(secret) → encWrite(secret,**none**)
writeBob(public) → encWrite(public,**none**)
writeEve(secret) → encWrite(secret,**none**)
writeEve(public) → encWrite(public,**none**)

No messages encrypted
Eve can read Alice's secret!

m_2:

writeBob(secret) → encWrite(secret,**keyX**)
writeBob(public) → encWrite(public,**keyX**)
writeEve(secret) → encWrite(secret,**keyX**)
writeEve(public) → encWrite(public,**keyX**)

Encrypt all messages
(safe but inefficient)

m_3:

writeBob(secret) → encWrite(secret,**keyX**)
writeBob(public) → encWrite(public,**none**)
writeEve(secret) → encWrite(secret,**keyX**)
writeEve(public) → encWrite(public,**none**)

Public messages
need no encryption

Verification and synthesis problems on mappings

- Verification: given application model P, platform model Q, mapping m, and some specification ϕ, check that the system $P \ ||_m Q$ satisfies ϕ.

- Synthesis: given P, Q and ϕ, find mapping m, such that $P \ ||_m Q$ satisfies ϕ.

Contributions

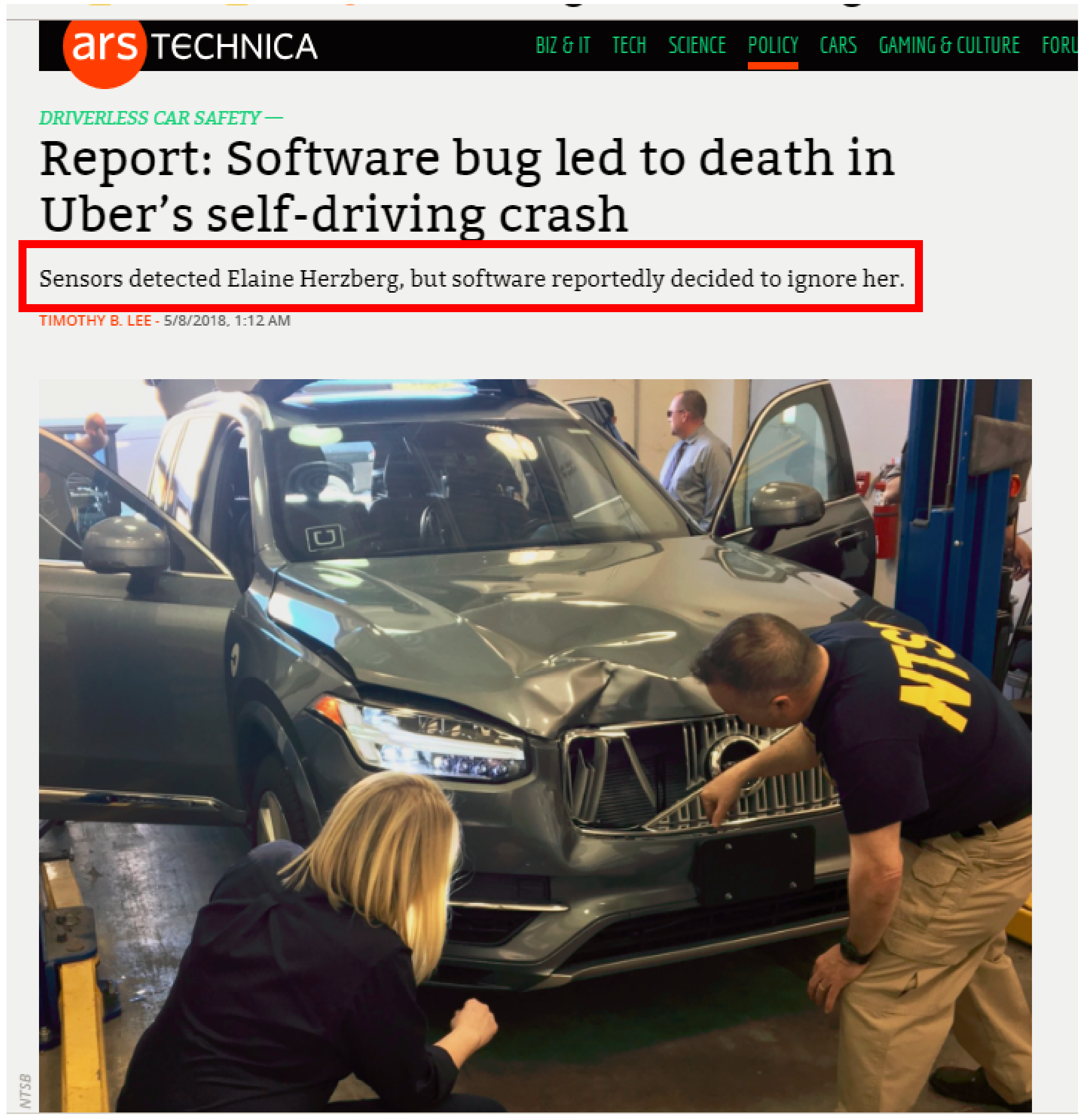
- Algorithm and tool for automated mapping synthesis:
  - Counter-example guided symbolic search over possible candidate mappings
- Real-world case studies: OAuth 2.0 and 1.0
  - Tool able to automatically synthesize correct mappings for both OAuth 2.0 and 1.0
  - Synthesized mappings describe mitigations to well-known attacks (e.g., session swapping, covert redirect, session fixation)
  - Several 1000s LOC of application and platform models: OAuth, HTTP server, HTTP browser, …

From Model-based to Data-driven and Model-based Design

Brave new world

Tempe, Arizona, March 18, 2018

- “Software designers face a basic tradeoff […]. If the software is programmed to be too cautious, the ride will be slow and jerky […]. Tuning the software in the opposite direction will produce a smooth ride most of the time—but at the risk that the software will occasionally ignore a real object. […] that's what happened in Tempe in March—and unfortunately the "real object" was a human being.”
- "There's a reason Uber would tune its system to be less cautious about objects around the car, […] It is trying to develop a self-driving car that is comfortable to ride in."


ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE FORU

DRIVERLESS CAR SAFETY —

Report: Software bug led to death in Uber’s self-driving crash

Sensors detected Elaine Herzberg, but software reportedly decided to ignore her.

TIMOTHY B. LEE - 5/8/2018, 1:12 AM

NTSB

New challenges and opportunities

- Can AI benefit from system design, and how?
- Can system design benefit from AI, and how?

Can AI benefit from system design?

- Yes.
- AI software is untestable.
- Formal verification of AI software is needed.

RAND CORPORATION

Driving to Safety

How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?

Nidhi Kalra, Susan M. Paddock

Key findings

- Autonomous vehicles would have to be driven hundreds of millions of miles and sometimes hundreds of billions of miles to demonstrate their reliability in terms of fatalities and injuries.
- Under even aggressive testing assumptions, existing fleets would take tens and sometimes hundreds of years

In the United States, roughly 32,000 people are killed and more than two million injured in crashes every year (Bureau of Transportation Statistics, 2015). U.S. motor vehicle crashes as a whole can pose economic and social costs of more than $800 billion in a single year (Blincoe et al., 2015). And, more than 90 percent of crashes are caused by human errors (National Highway Traffic Safety Administration, 2015)—such as driving too fast and misjudging other drivers' behaviors, as well as alcohol impairment, distraction, and fatigue.

Can system design benefit from AI?

- Yes.

- Data-driven and Model-based Design (DMD)

Data-driven and Model-based Design – motivation and goals

- Combine the best of both worlds:
  - Trial-and-error
  - Model-based design

- Leverage advances in AI (machine learning, data science, …) to improve system design methods.

- Complement existing AI methods by developing new techniques developed specifically for system design.

Example: combining controller synthesis and learning

Joint work with Rajeev Alur, Christos Stergiou et al (UPenn)

Sponsors: NSF Expeditions ExCAPE

Motivation: distributed protocols

- Notoriously hard to get right

Can we **synthesize** such protocols **automatically**?

COMMUNICATIONS OF THE ACM

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRACTICE | CAREERS | ARCHIVE | VIDEOS

Home / Magazine Archive / April 2015 (Vol. 58, No. 4) / How Amazon Web Services Uses Formal Methods / Full Text

CONTRIBUTED ARTICLES

How Amazon Web Services Uses Formal Methods

(to model and verify distributed protocols)

By Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Communications of the ACM, Vol. 58 No. 4, Pages 66-73
10.1145/2699417
Comments (1)

VIEW AS: SHARE:

Since 2011, engineers at Amazon Web Services (AWS) have use formal specification and model checking to help solve difficult design problems in critical systems. Here, we describe our motivation and experience, what has worked well in our proble domain, and what has not. When discussing personal experien we refer to the authors by their initials.

At AWS we strive to build services that are simple for customer to use. External simplicity is built on a hidden substrate of

Key Insights

- Formal methods find bugs in system designs that cannot be found through any other technique we know of.
- Formal methods are surprisingly feasible for mainstream software development and give good return on investment.
- At Amazon, formal methods are routinely applied to the design of complex real-world software, including public cloud services.

Verification and synthesis in a nutshell

- Verification:
  1. Design system "by hand": S
  2. State system requirements: ϕ
  3. Check: does S satisfy ϕ ?

- Synthesis (ideally):
  1. State system requirements: ϕ
  2. Generate automatically system S that satisfies ϕ by construction.

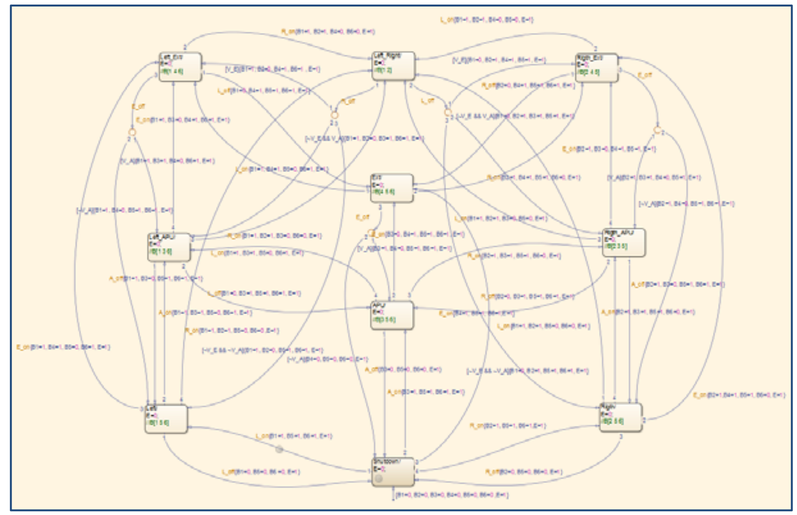
State of the art synthesis

- From formal specs to discrete controllers:

```
#Assumptions
(gl_healthy & gr_healthy & al_healthy & ar_healthy)
[](gl_healthy | gr_healthy | al_healthy | ar_healthy)
[](!gl_healthy -> X(!gl_healthy) )
[](!gr_healthy -> X(!gr_healthy) )
[](!al_healthy -> X(!al_healthy) )
[](!ar_healthy -> X(!ar_healthy) )

#Guarantees
(!c1 & !c2 & !c3 & !c4 & !c5 & !c6 & !c7 & !c8 & !c9 & !c10 &
!c11 & !c12 & !c13)
[](X(c7) & X(c8) & X(c11) & X(c12) & X(c13))
[](!(c2 & c3))
[](!(c1 & c5 & (al_healthy | ar_healthy)))
[](!(c4 & c6 & (al_healthy | ar_healthy)))
[]((X(gl_healthy) & X(gr_healthy) ) -> X(!c2) & X(!c3) &
X(!c9) & X(!c10))
[]((X(!gl_healthy) & X(!gr_healthy) ) -> X(c9) & X(c10))
…
```

Specification (temporal logic formulas)

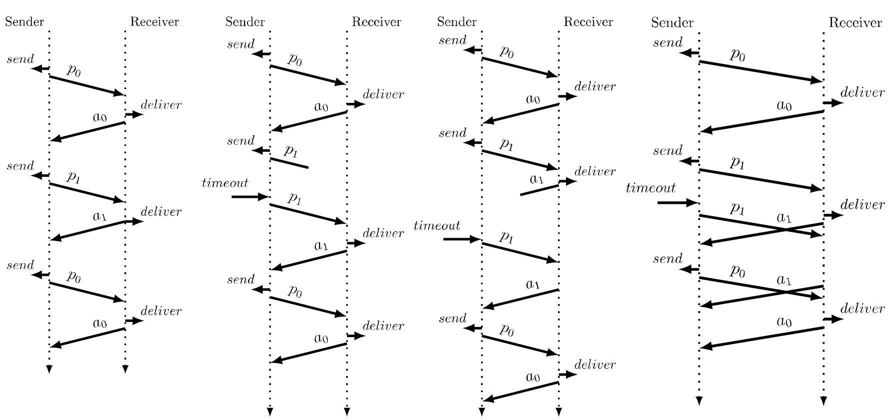


Controller (state machine)

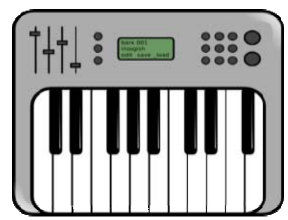
- Limitations:
  - Scalability (writing full specs & synthesizing from them)
  - Not applicable to distributed protocols (undecidable)

Synthesis of Distributed Protocols from Scenarios and Requirements

- Idea: **combine requirements** + example **scenarios**



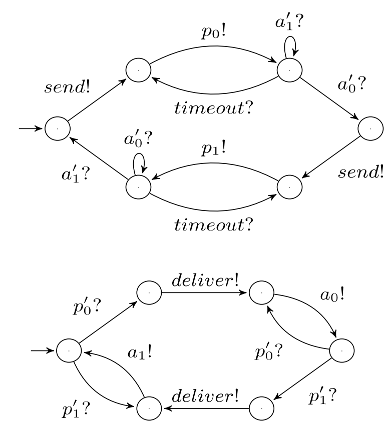
example scenarios



Synthesis tool

formal requirements (safety, liveness, deadlock-freedom, …)

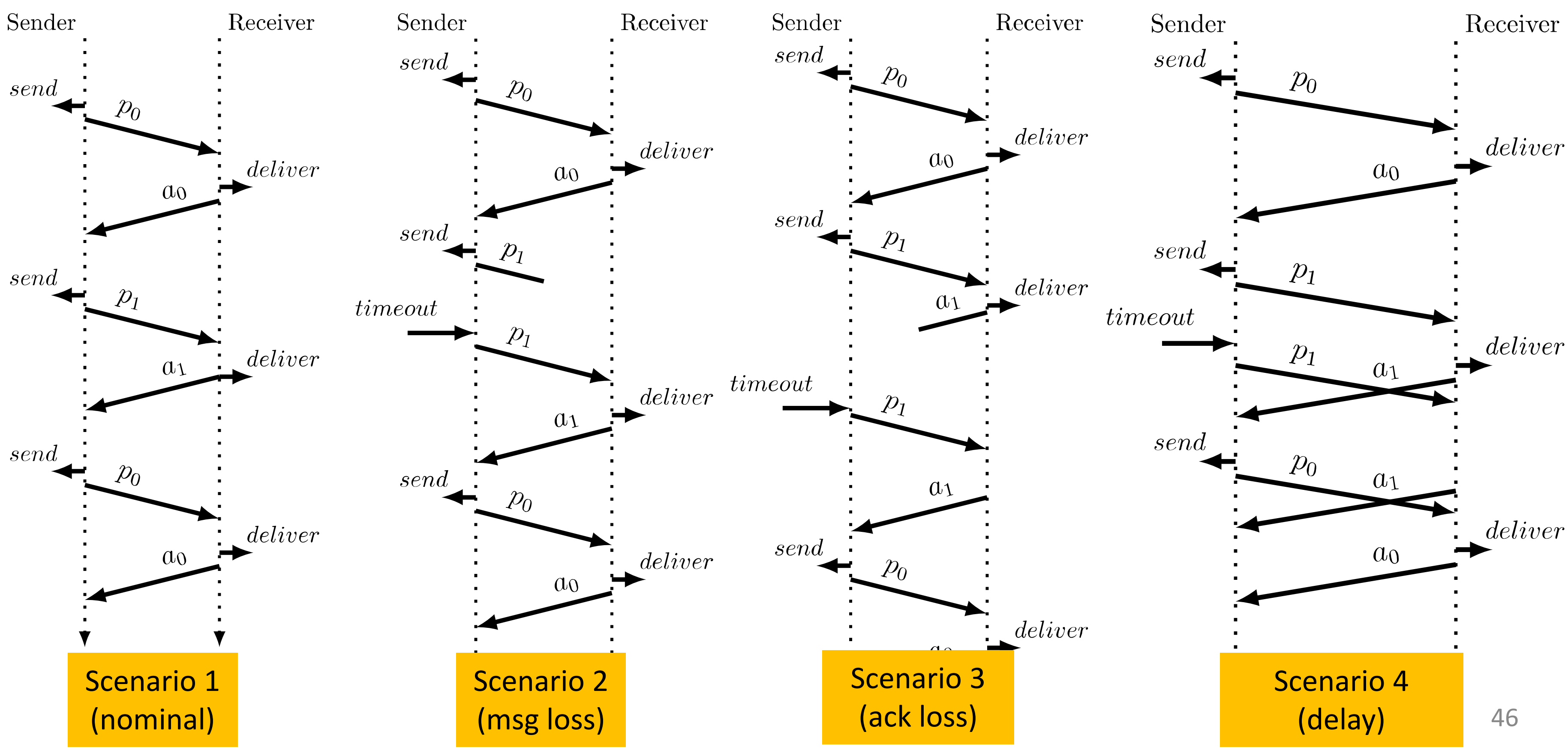
These are typically not complete specs!



synthesized protocol (state machines)

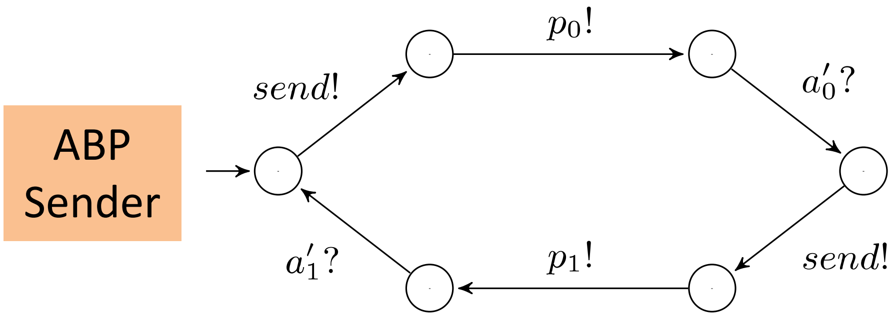
Scenarios: message sequence charts

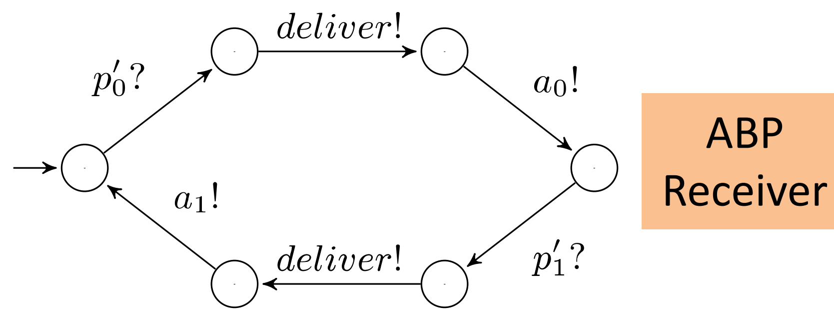
- Describe what the protocol must do in **some** cases
- Intuitive language ⇒ good for the designer
- Only a few scenarios required (1-10)



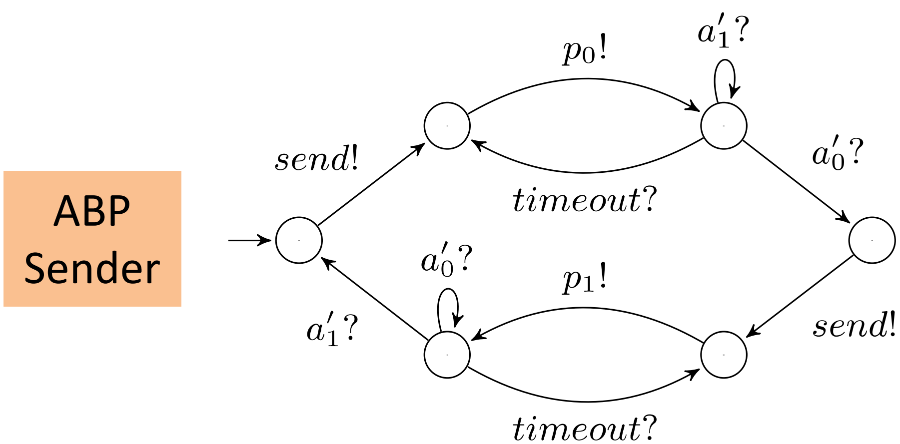
Synthesis becomes a **completion** problem

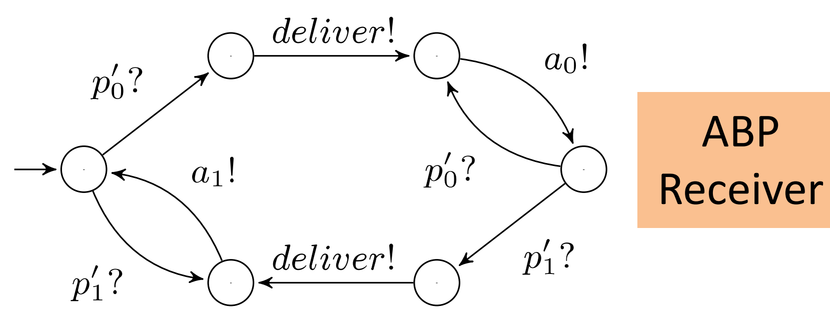
Incomplete automata learned from first scenario:





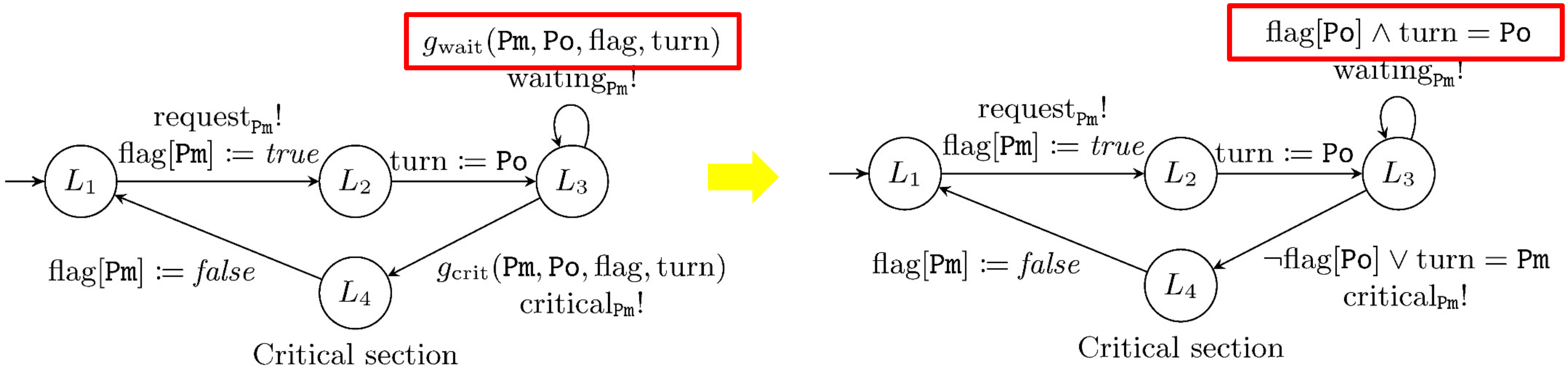
Automatically completed automata:





Results

- Able to synthesize the distributed Alternating Bit Protocol (ABP) and other simple finite-state protocols (cache coherence, consensus, …) fully automatically [HVC'14, ACM SIGACT'17].
- Towards industrial-level protocols described as **extended state machines** [CAV'15].



Algorithmic technique: counter-example guided completion of (extended) state machines

- Completion of incomplete machines: find missing transitions, guards, assignments, etc.

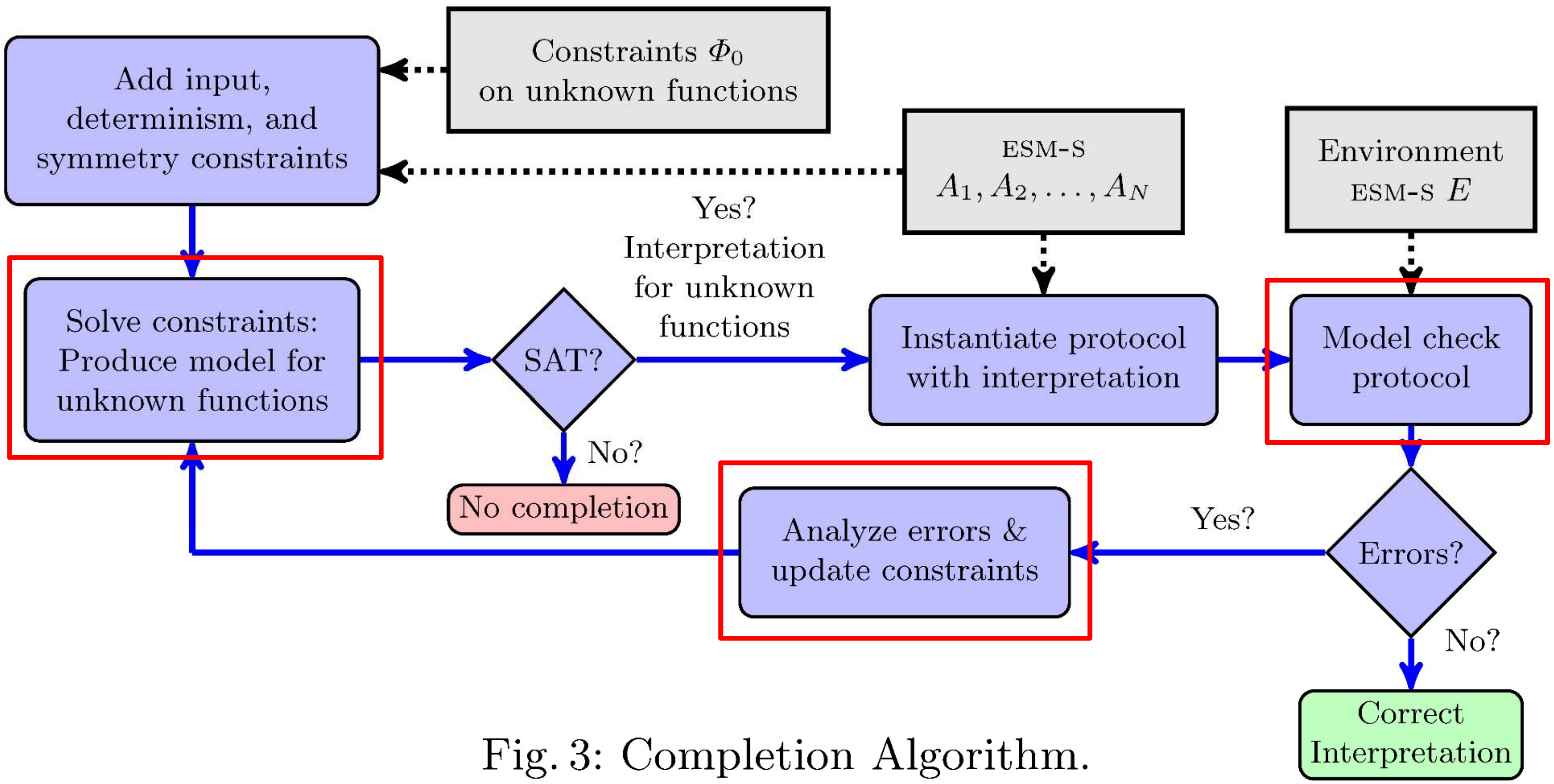


Fig. 3: Completion Algorithm.

Combining synthesis with learning

- Synthesis: given specification ϕ, find system S, such that $S \vDash \phi$
- Learning: given set of examples E, find system S, such that S is consistent with E and "generalizes well" …
- Synthesis from spec + examples: given set of examples E and specification ϕ, find system S, such that S is consistent with E and $S \vDash \phi$
  – Key advantage: ϕ guides the generalization!

CONCLUSIONS

The science of system design

- Theory and tools that help us make better predictions about the systems we build.
- Formal modeling, verification, synthesis, …
  - A.k.a. “formal methods”.
- Broad spectrum of interesting research problems (theory and practice).
  - Increasingly mature for education.
- Increasingly popular in the industry.
- New opportunities: data, examples, learning!

Thank you

Questions?