The Science of Software and System Design

Stavros Tripakis
Northeastern University

INDIN, 24 July 2019

Science = knowledge that helps us make predictions

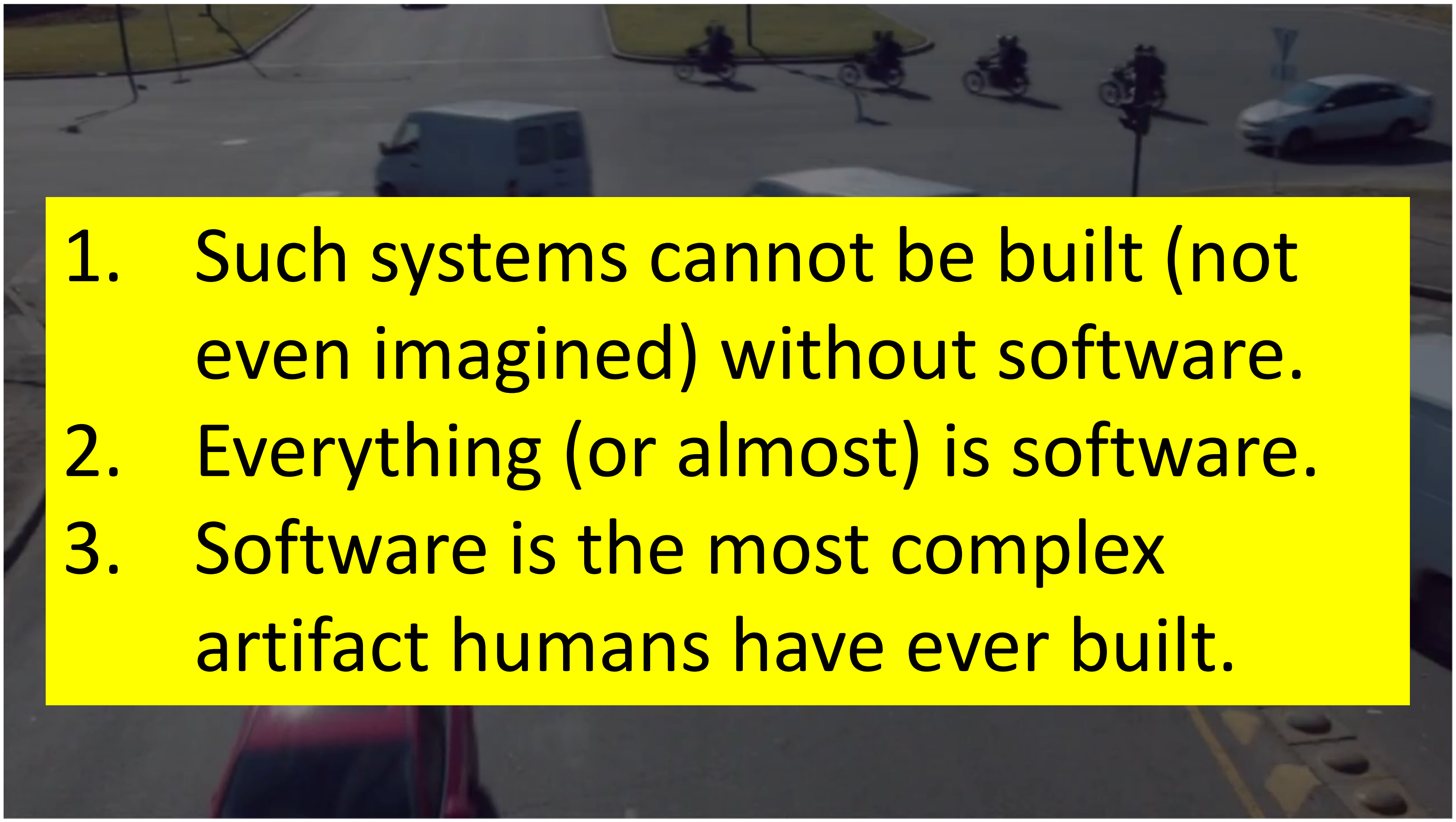
- **Science of software**: what predictions can we make about the programs we write?
  - My program terminates
  - My program doesn't throw an exception
  - My program satisfies properties X, Y, Z, …

- **Science of systems** (safety-critical, real-time, embedded, cyber-physical, secure, … systems)
  - Many specialized techniques (e.g., for linear systems, …)
  - **Thesis: same fundamental methods as for software**

Cyber-physical systems: present

Autonomous car driving through red light

‘Cyber-physical’ systems: future



1. Such systems cannot be built (not even imagined) without software.
2. Everything (or almost) is software.
3. Software is the most complex artifact humans have ever built.

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

Software and complexity

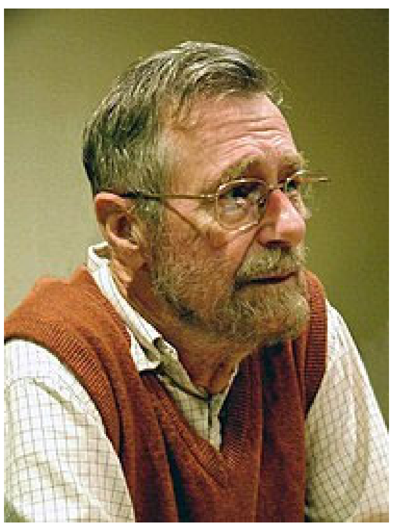
```
int x := input an integer number > 1;

while x > 1 {
  if x is even
      x := x / 2;
  else
      x := 3*x + 1;
}
```

Run starting at 31: 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412 206 103 310 155 466 233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668 334 167 502 251 754 377 1132 566 283 850 425 1276 638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288 3644 1822 911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433 1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2

Collatz conjecture:
the program terminates for every input.
Open problem in mathematics.

Basic software and system design methods

- **Testing** (trial and error)
- **Proving** (specification/verification-based design, model-based design)



"Testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra, 1970]

Formal verification: a successful and practical approach

COMMUNICATIONS OF THE ACM

Search

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) ha… formal specification and model checking to help solve dif… design problems in critical systems. Here, we describe ou… motivation and experience, what has worked well in our … domain, and what has not. When discussing personal exp… we refer to the authors by their initials.

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

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.

Boeing 737 Max 8 accidents

- 2 accidents within 5 months – 346 deaths
- 737 Max 8 planes grounded world-wide since March 2019
- Control system rather than pilot errors
- Dubious business and certification practices

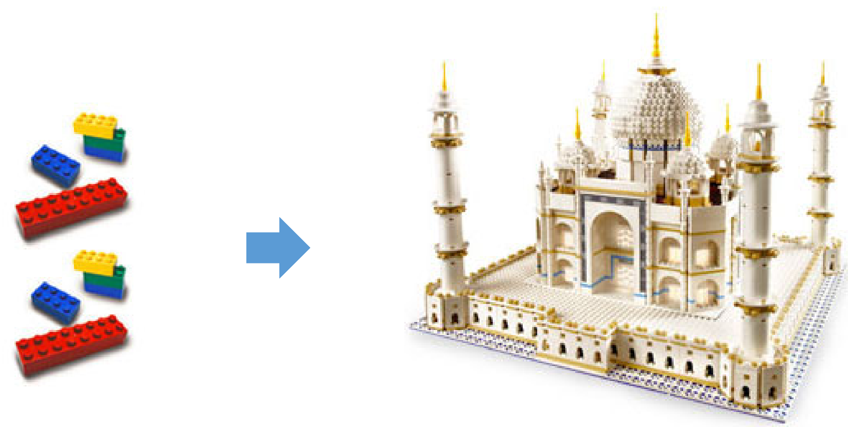
Recent work topics

- The Refinement Calculus of Reactive Systems
- Synthesis of distributed protocols – with connections to learning
- Synthesis of platform mappings – with applications to security
- Multi-view modeling

The Refinement Calculus of Reactive Systems (RCRS)

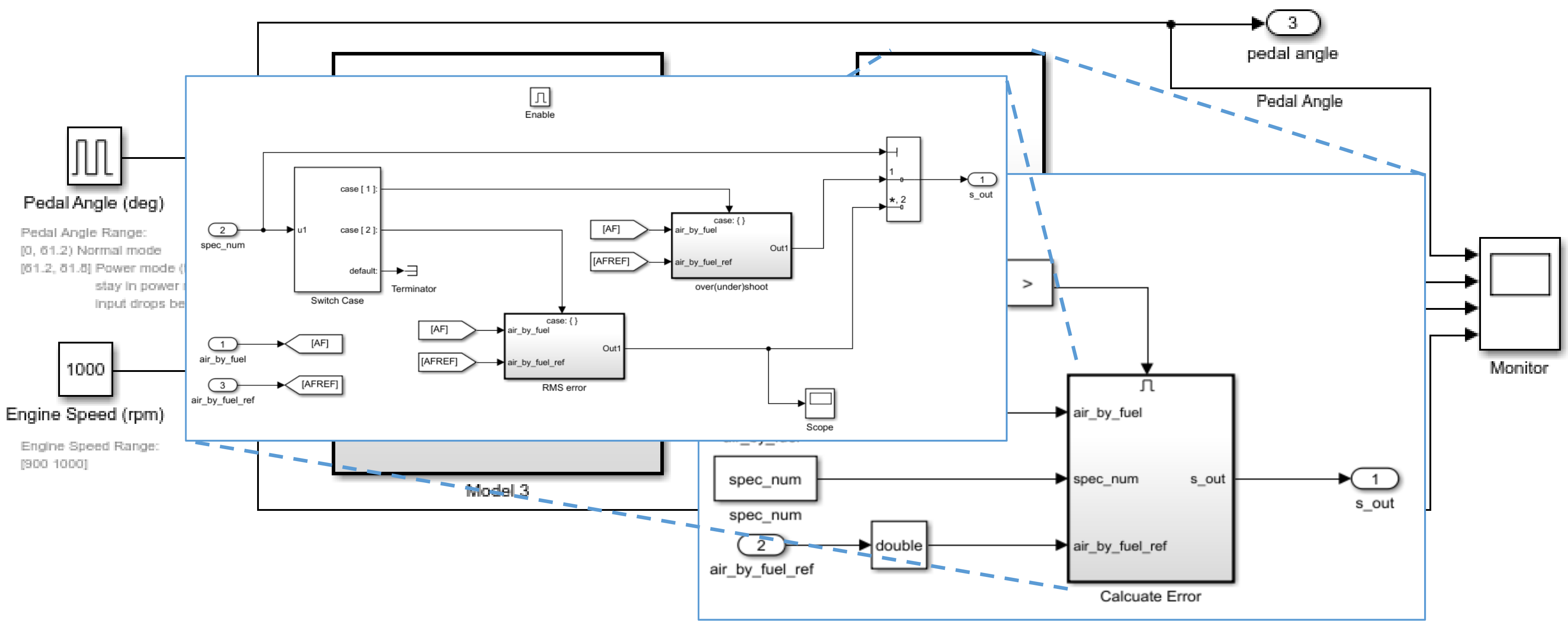
Joint work with Viorel Preoteasa and Iulia Dragomir (Aalto)

Sponsors: Academy of Finland and NSF CPS Breakthrough



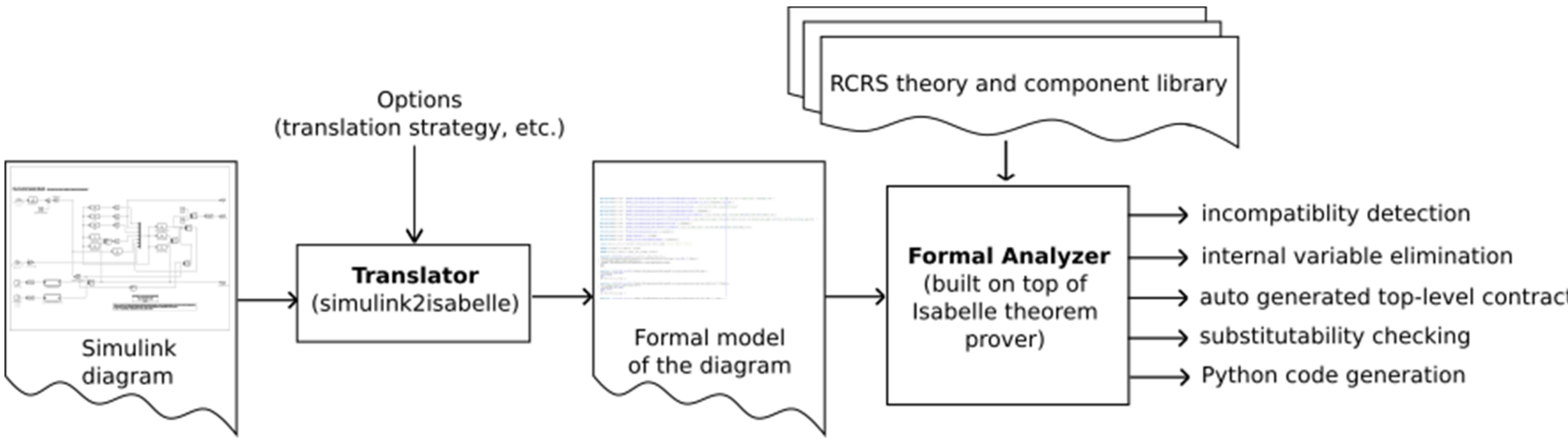
Modeling cyber-physical systems compositionally

- Simulink: a de-facto standard



Distinguished Artifact Award
TACAS 2018

RCRS: theory & tool for compositional formal analysis of Simulink models



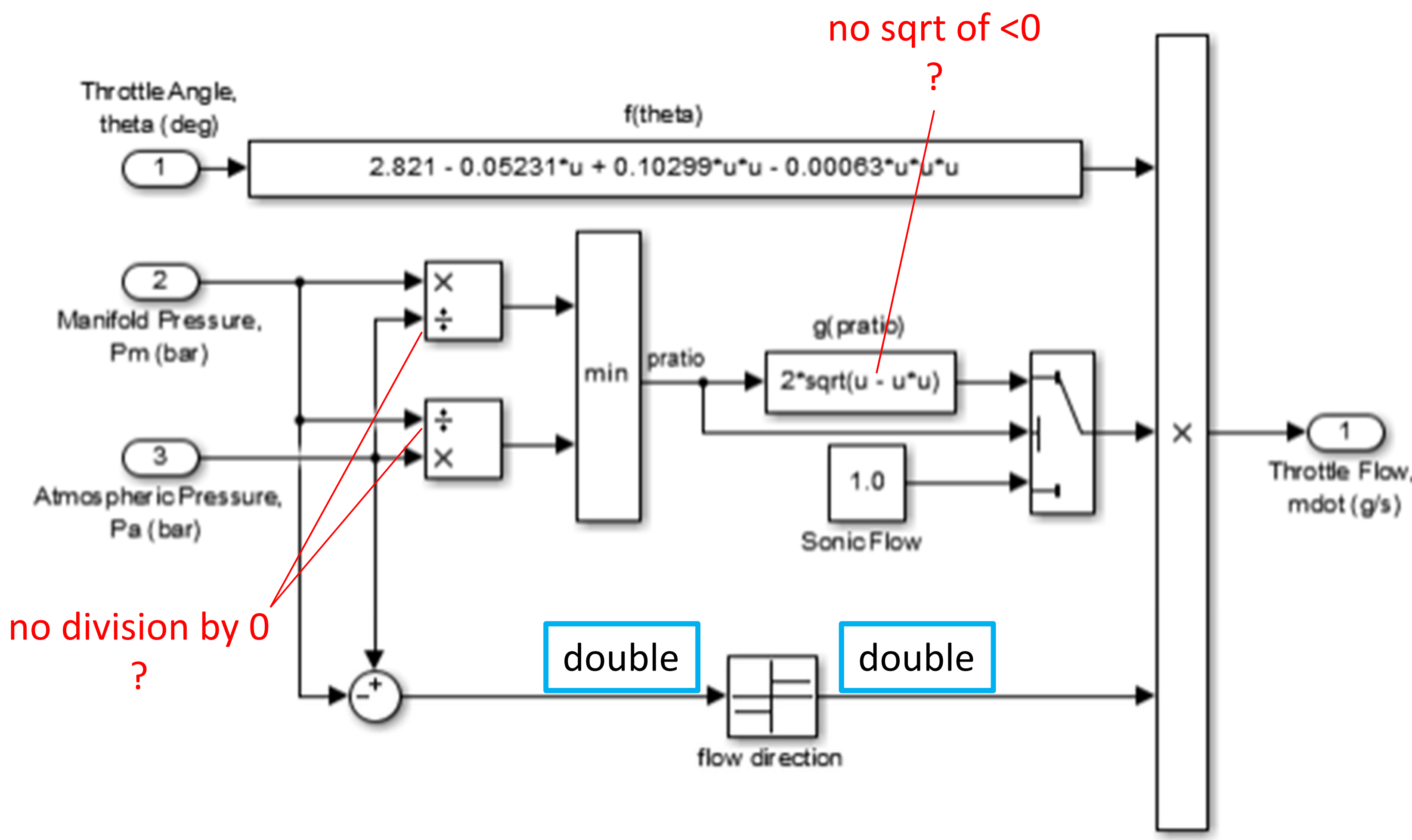
Downloadable from http://rcrs.cs.aalto.fi/

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

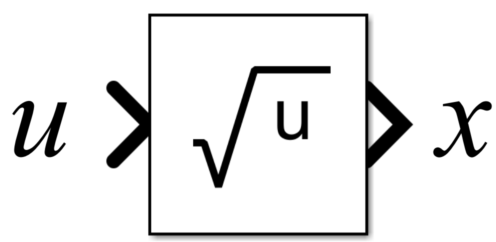
- Refinement calculus of reactive systems [EMSOFT'14]
  - Richer semantics: predicate and property transformers
  - Can handle both safety and liveness properties
  - Entirely formalized in Isabelle theorem prover - 30k lines of Isabelle code

Static ('compile-time') analysis



Based on Simulink Demo, Copyright 1990-2010 The MathWorks, Inc.

Simulink square root modeled with RCRS contracts



Sqrt

```
double -> double
```

Simulink type

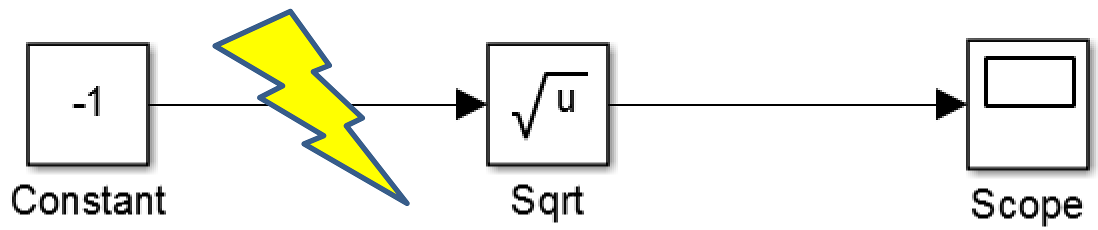
$$u \geq 0 \rightarrow x = \sqrt{u}$$

RCRS contract:
input-receptive

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

RCRS contract:
non-input-receptive

Catching incompatibilities statically

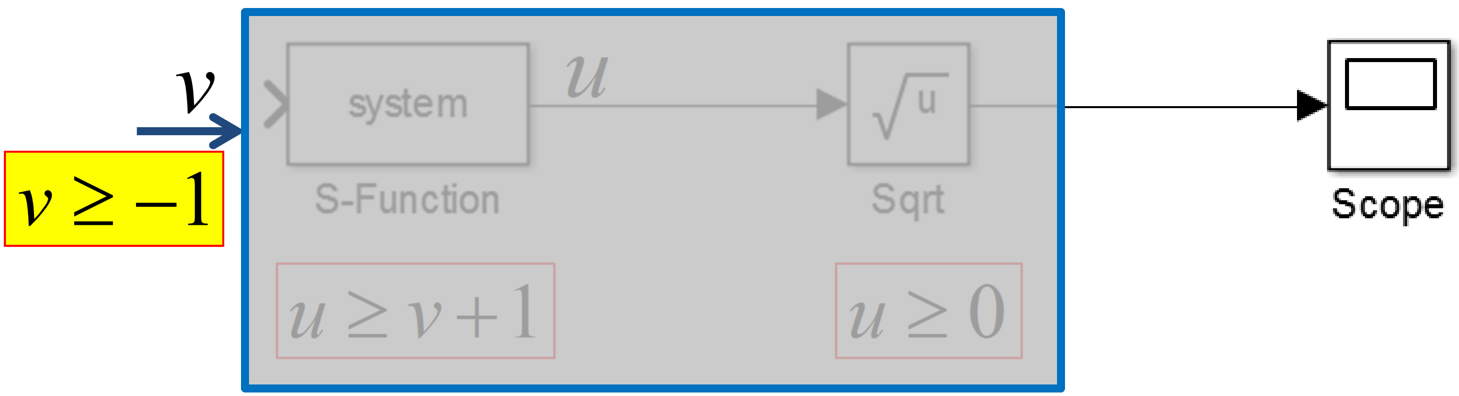


$u = -1$

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

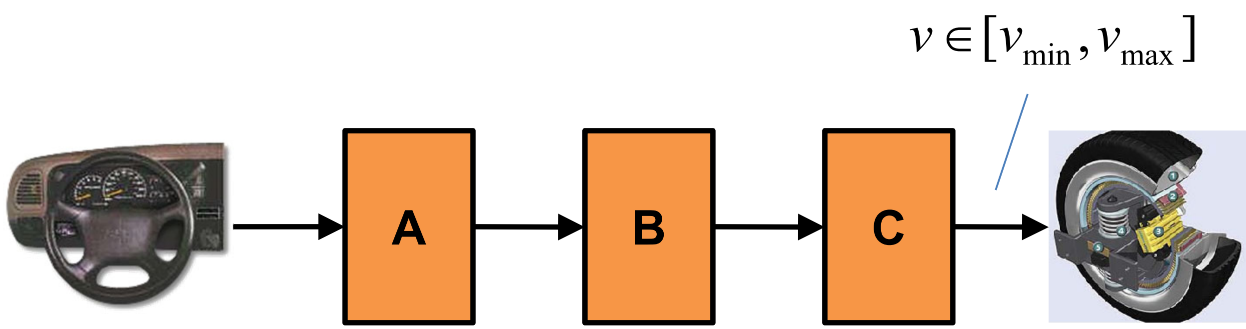
caught by taking the conjunction of the two formulas
and checking satisfiability

Inferring new contracts automatically



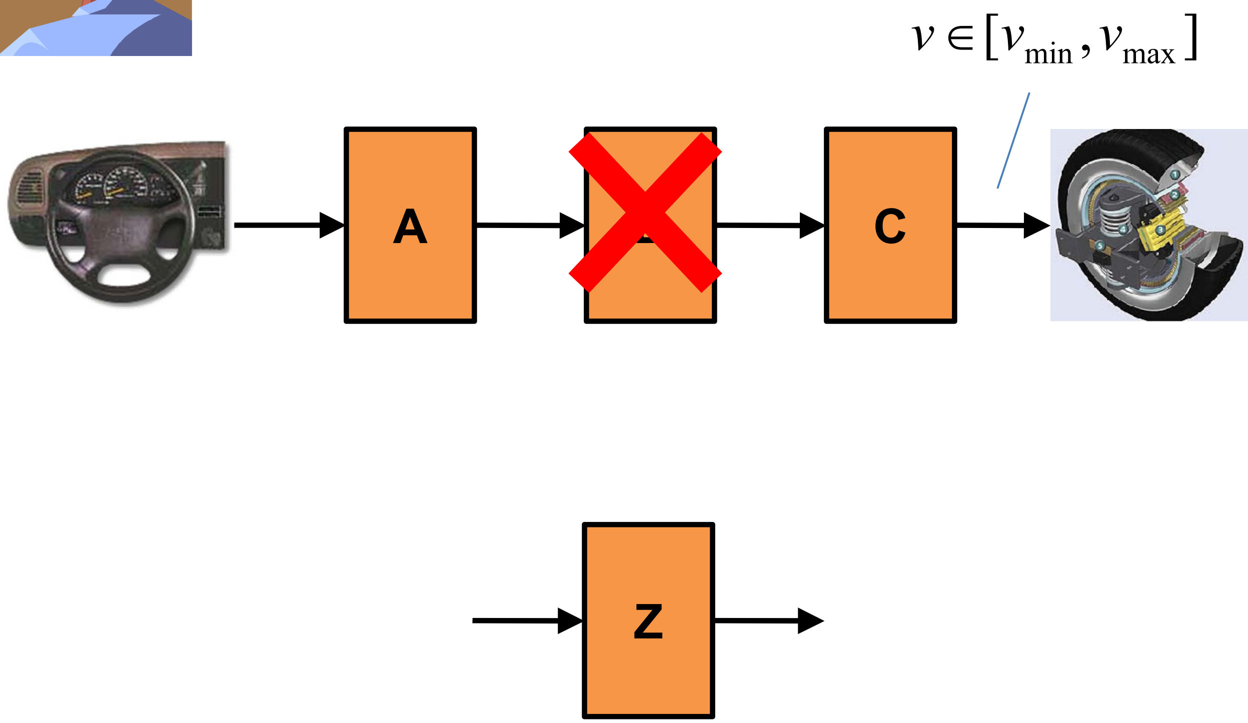
Software evolution with refinement

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



Software evolution with refinement

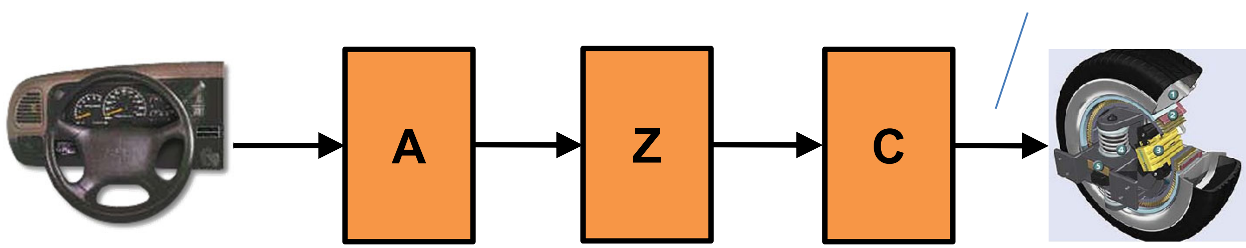
Suppose we want to replace B with Z:



Software evolution with refinement

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

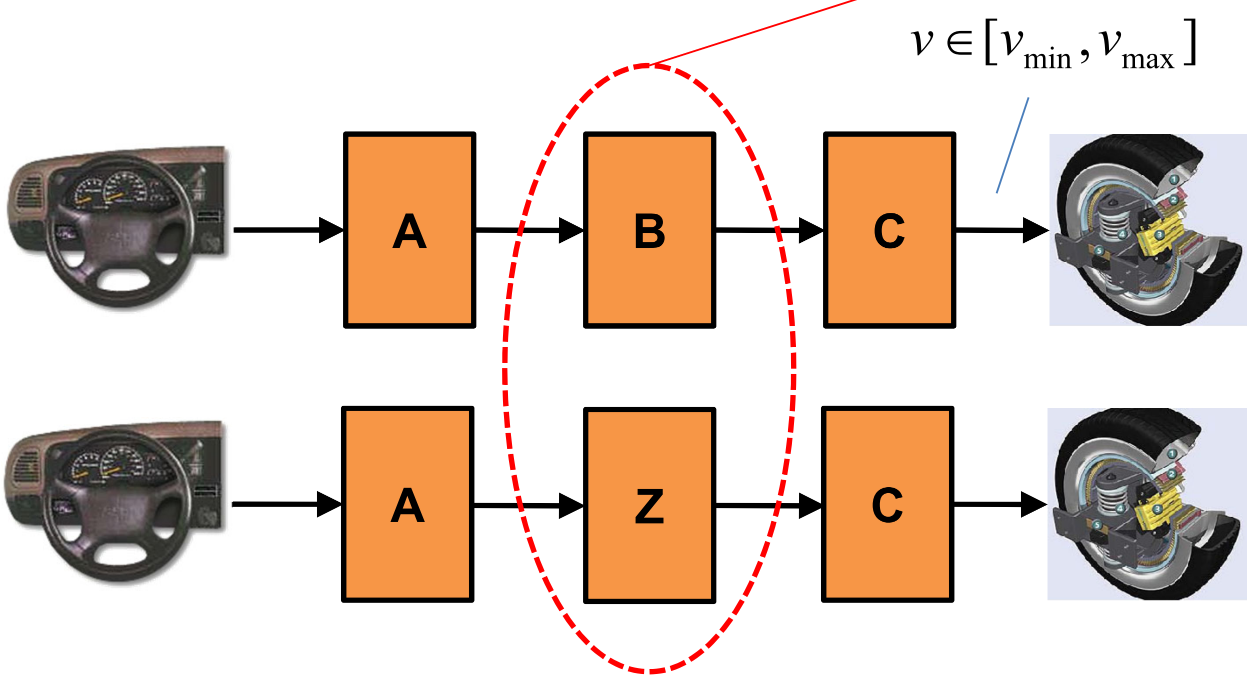
$v \in [v_{\min}, v_{\max}]$



Software evolution with refinement

Compositional theories like RCRS offer **local** verification methods.

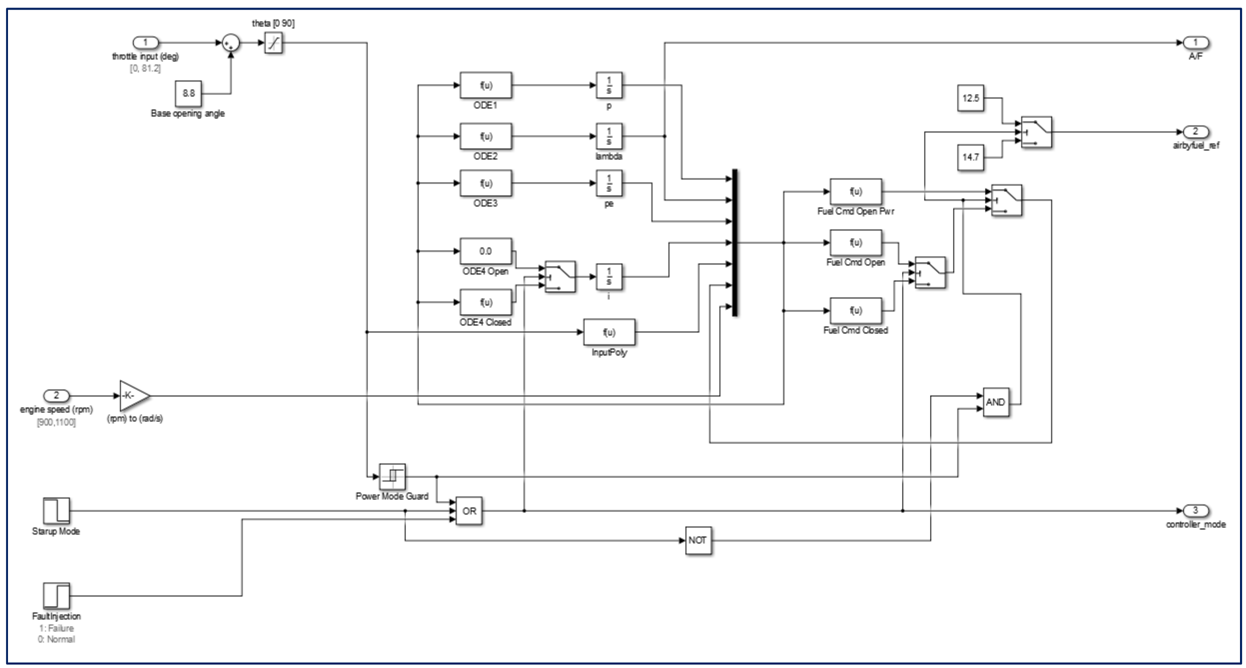
$Z \leq B$: Z *refines* B (local check)



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



Automatic synthesis of distributed protocols

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

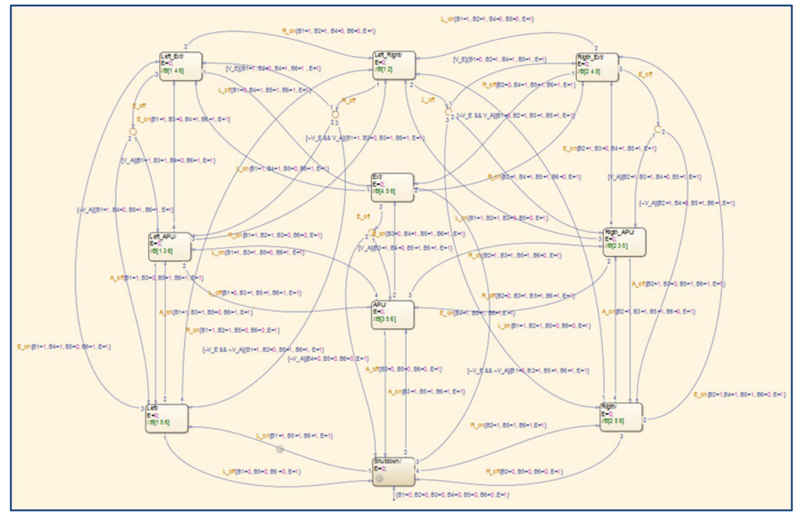
- Verification:
  1. Design system "by hand": S
  2. State system requirements: ϕ
  3. Check: does S satisfy ϕ ?
- Synthesis (ideally):
  1. State system requirements: ϕ
  2. Generate automatically system S that satisfies ϕ by construction.

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)

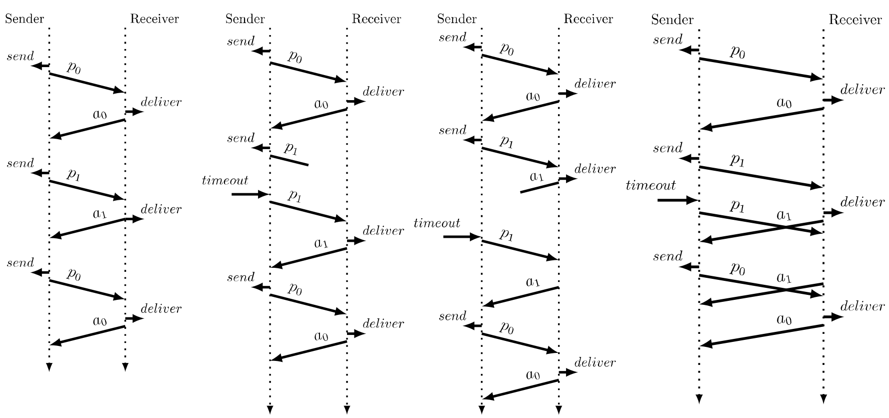


Controller (state machine)

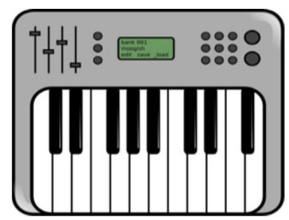
- Limitations:
  - Scalability (writing full specs & synthesizing from them)
  - Not applicable to distributed protocols (undecidable)

Synthesis of Distributed Protocols from Scenarios and Requirements

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

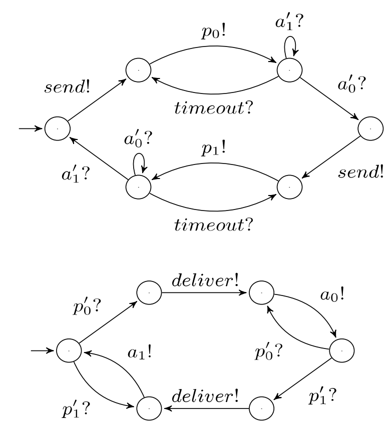


example scenarios



Synthesis tool

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

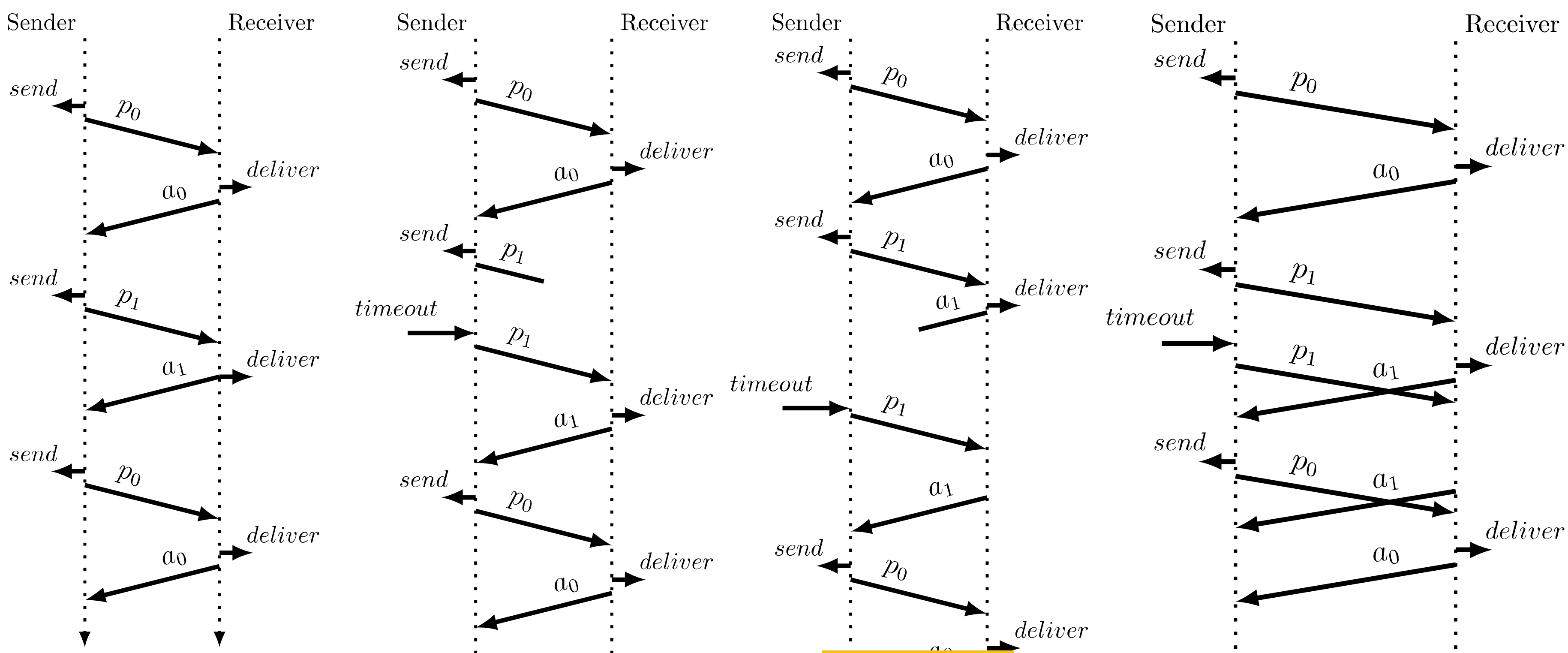


synthesized
protocol
(state machines)

These are typically not complete specs!

Scenarios: message sequence charts

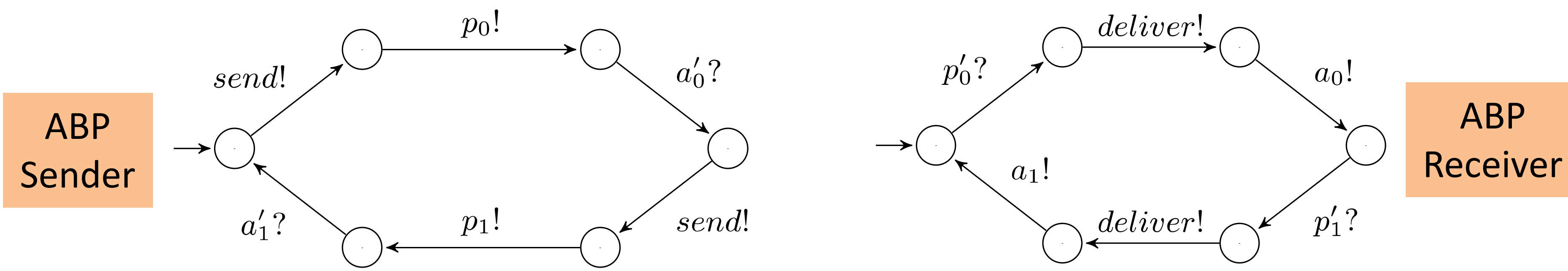
- Describe what the protocol must do in **some** cases
- Intuitive language ⇒ good for the designer
- Only a few scenarios required (1-10)



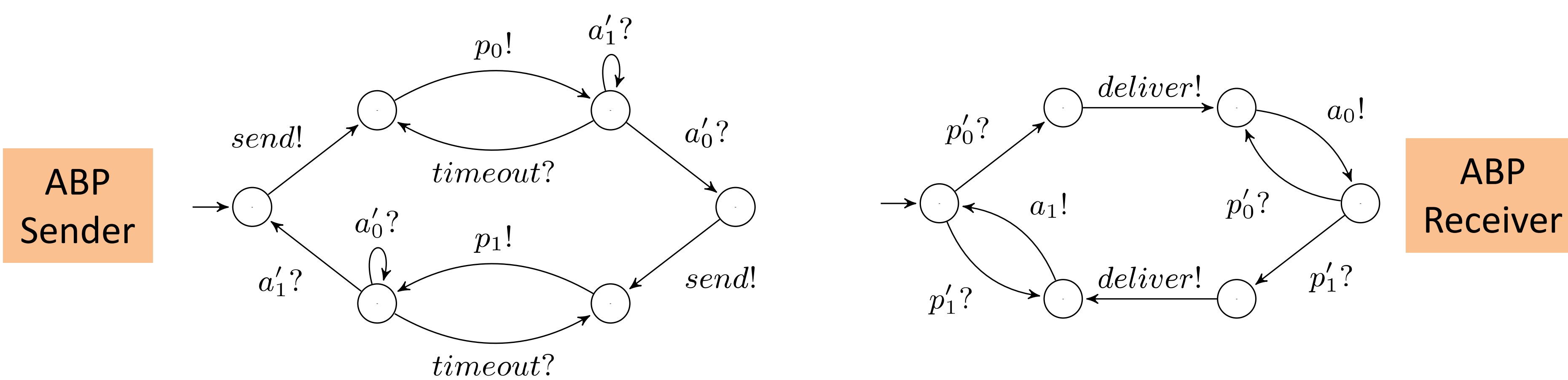
Scenario 1 (nominal) | Scenario 2 (msg loss) | Scenario 3 (ack loss) | Scenario 4 (delay)

Synthesis becomes a **completion** problem

Incomplete automata learned from first scenario:

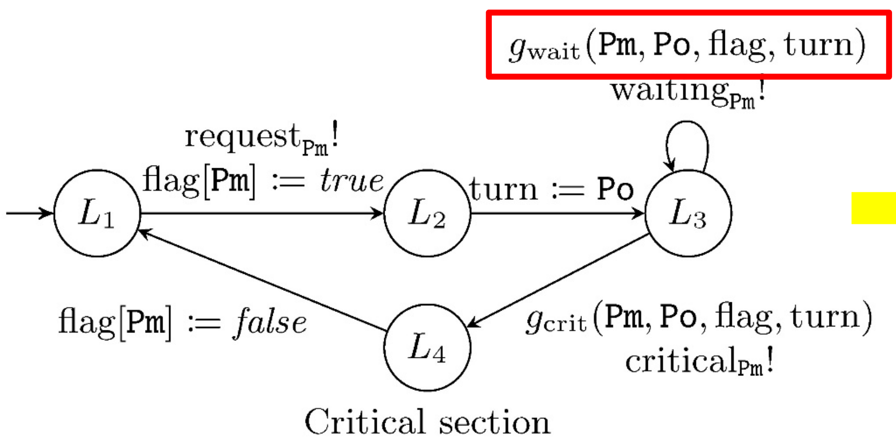


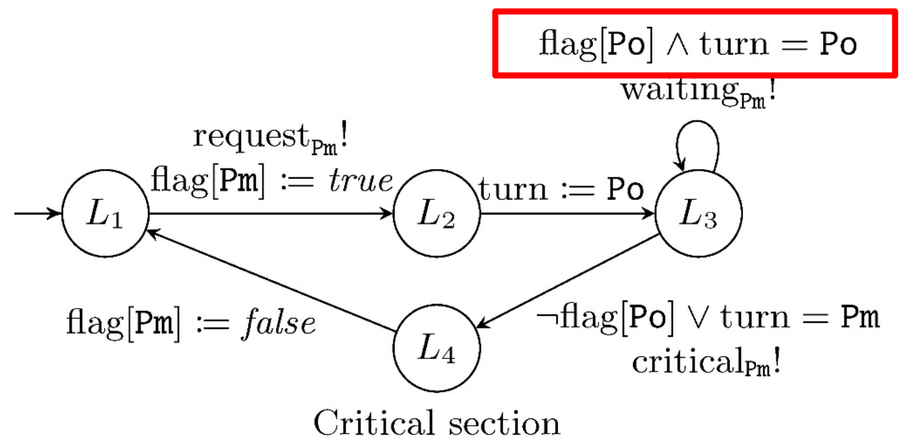
Automatically completed automata:



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.

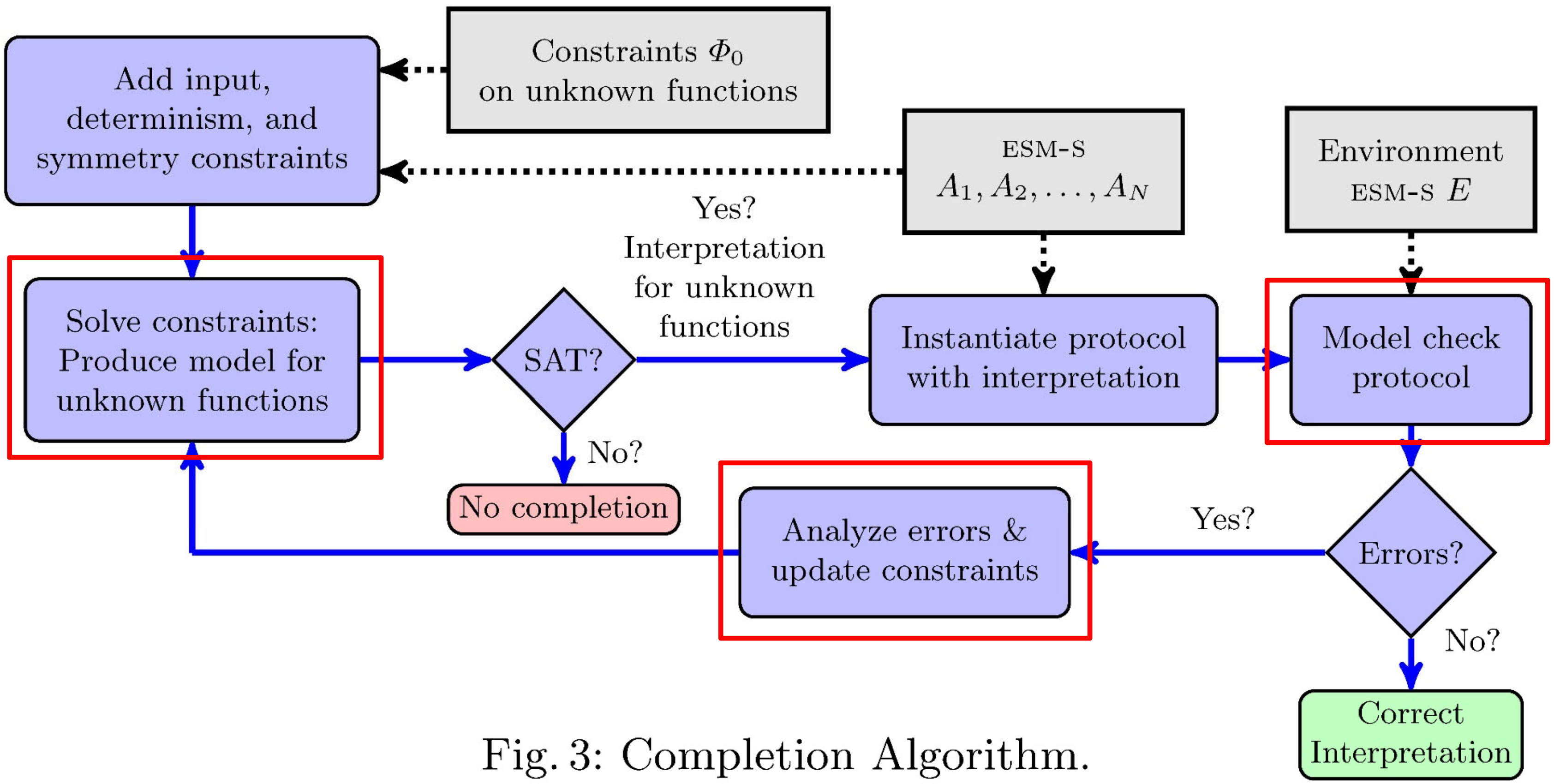


Fig. 3: Completion Algorithm.

Synthesis of platform mappings with applications to security

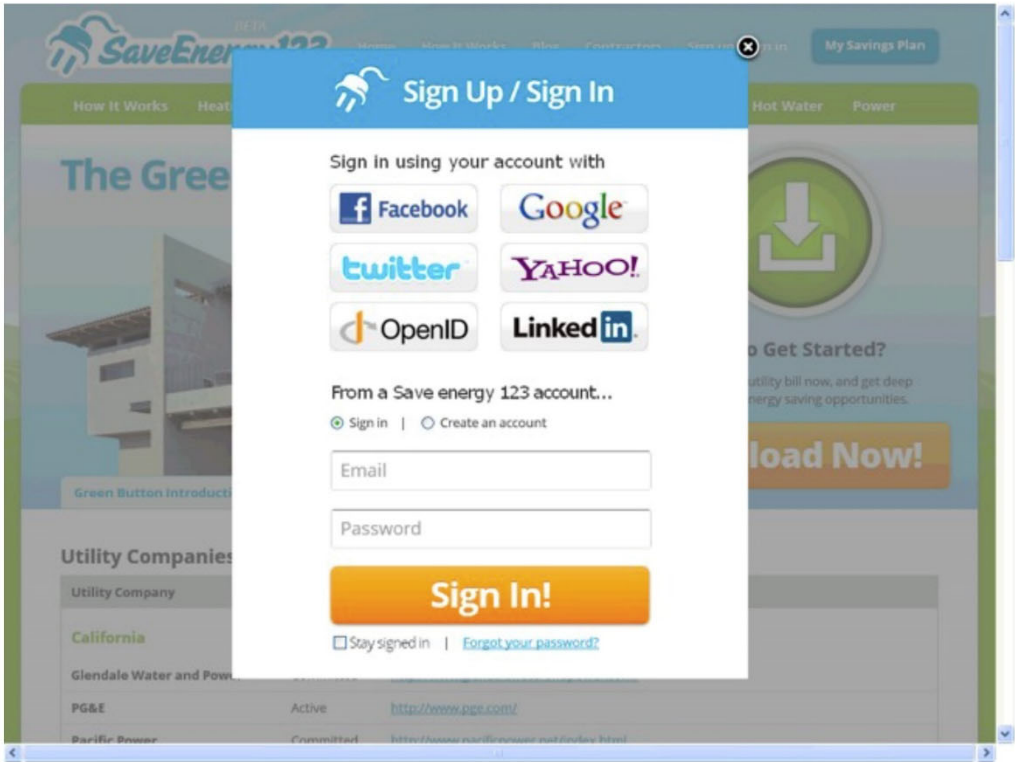
Joint work with Eunsuk Kang (NSF ExCAPE project), and Stephane Lafortune (UMichigan)

Sponsors: NSF SaTC program

Thanks to Eunsuk Kang for several slides

Motivation: security

Third-Party Authentication



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

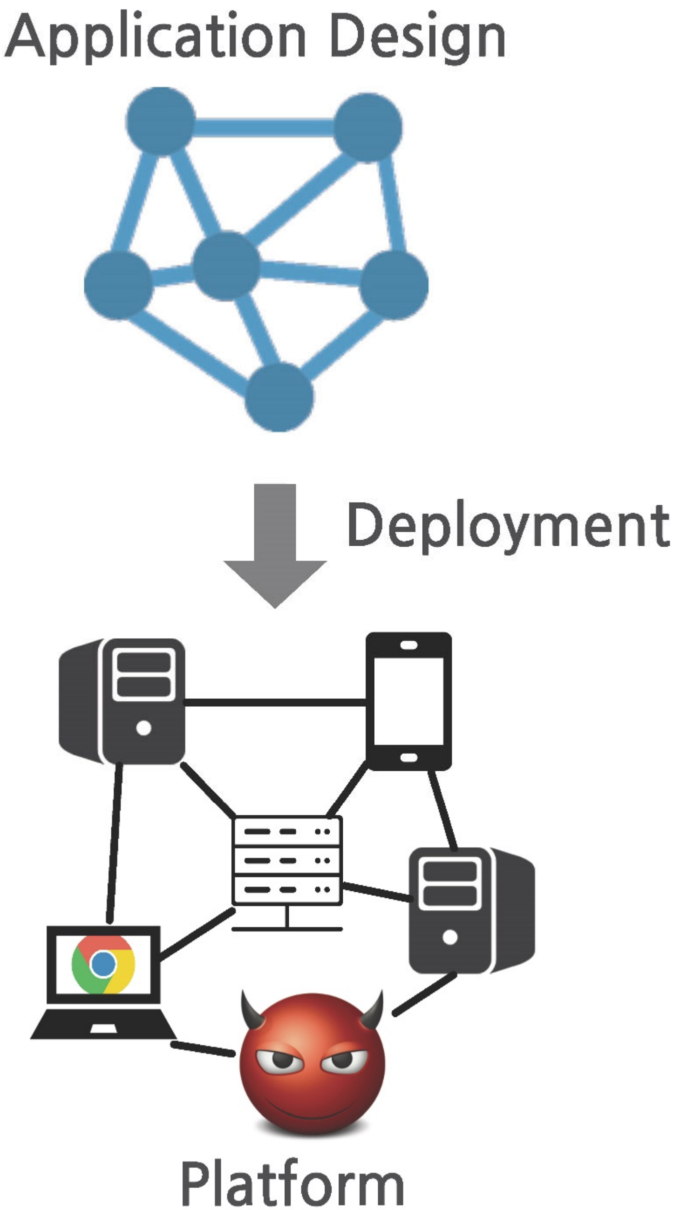
Motivation: security

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



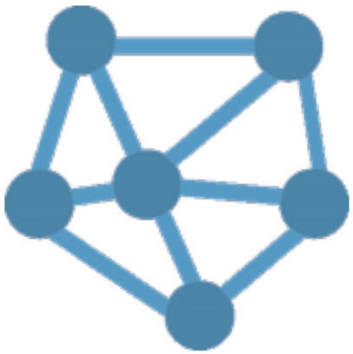
Designers think at high-level
Protocols, APIs, workflows,
use cases, etc.,
Ignore irrelevant details

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

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 [CAV 2019]

- 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, …

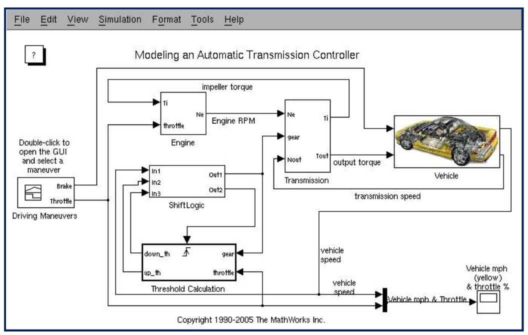
Multi-view modeling

Joint work with Jan Reineke (Saarland), Christos Stergiou (now at Google), and Maria Pittou (ex PhD student)

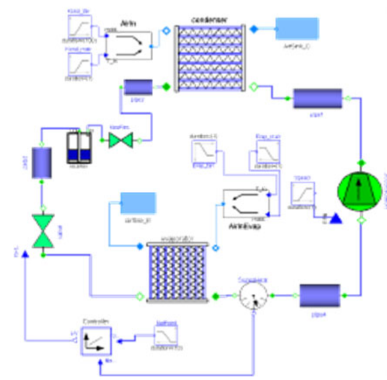
Part of NSF Project COSMOI

Multi-View Modeling

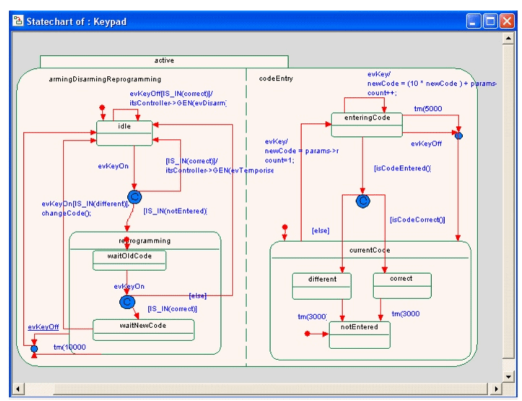
Complex system -> many stakeholders -> many design teams -> many viewpoints -> many perspectives -> many models = views



Low-level controllers Simulink



Supervisory controllers Rhapsody/ SysML

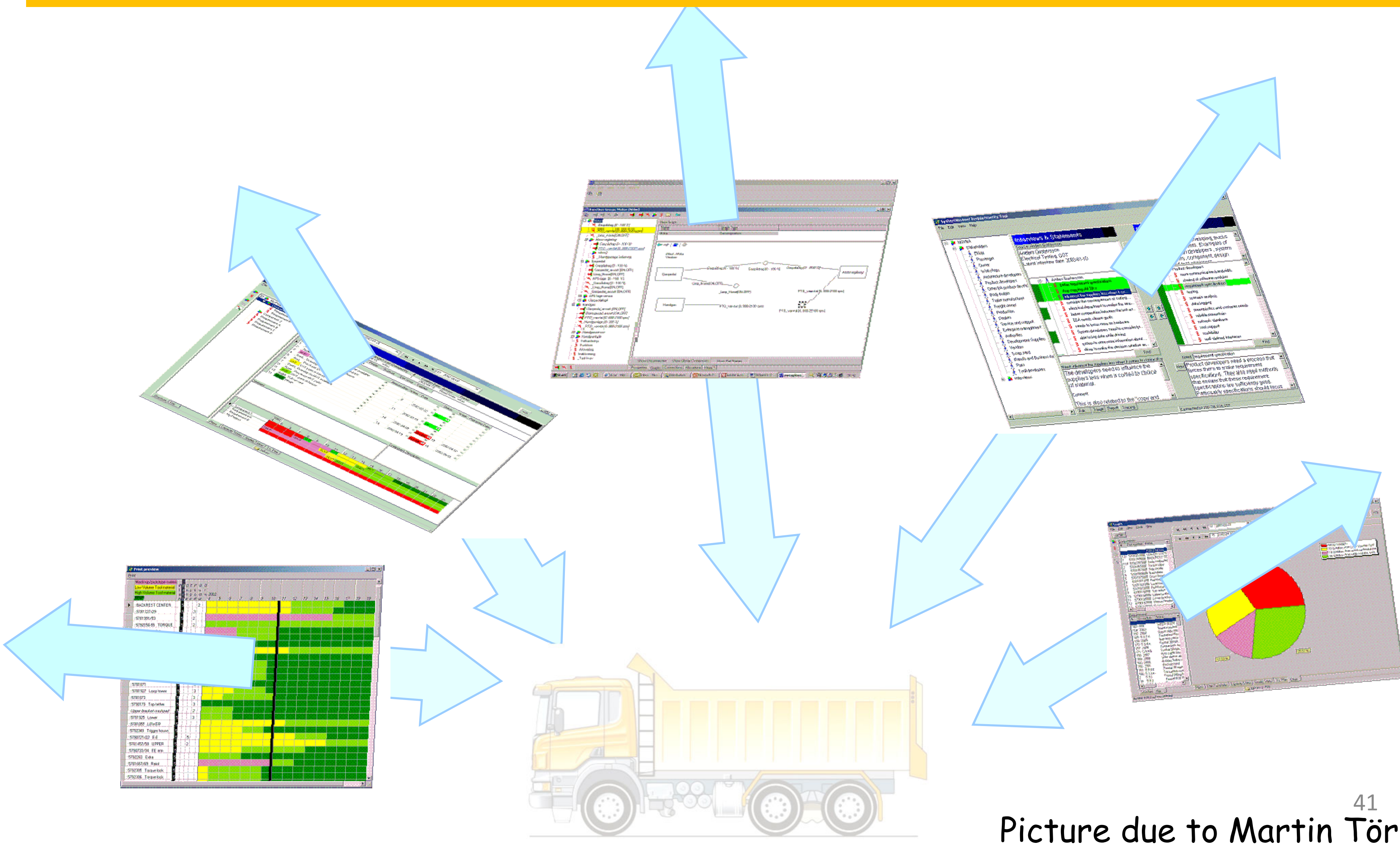


Physical dynamics Modelica

Problem: View Consistency

Partially overlapping content -> potential for **contradictions**

Industry: “system **integration** is the biggest issue”

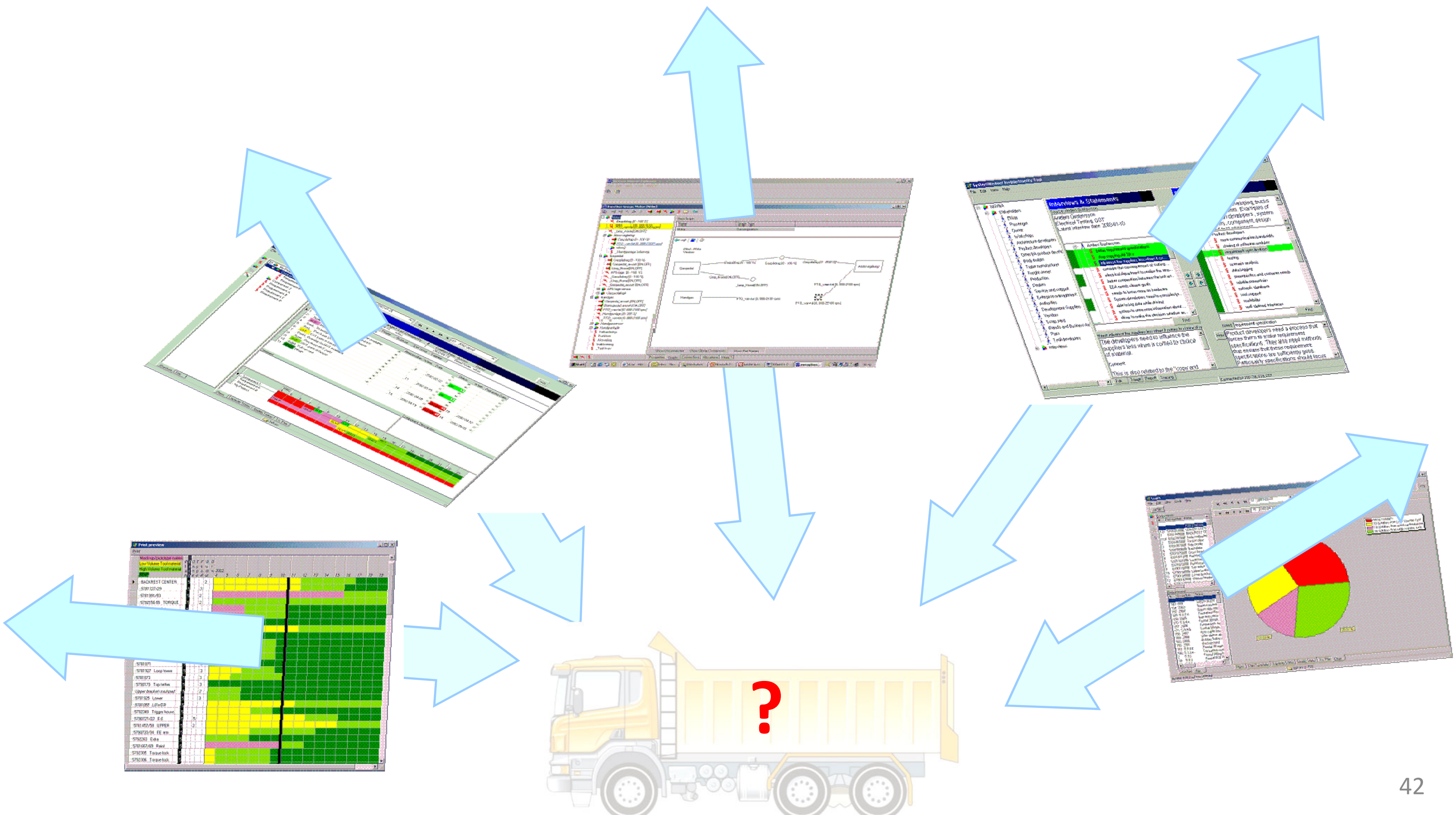


Picture due to Martin Törngren

What is view consistency, formally?

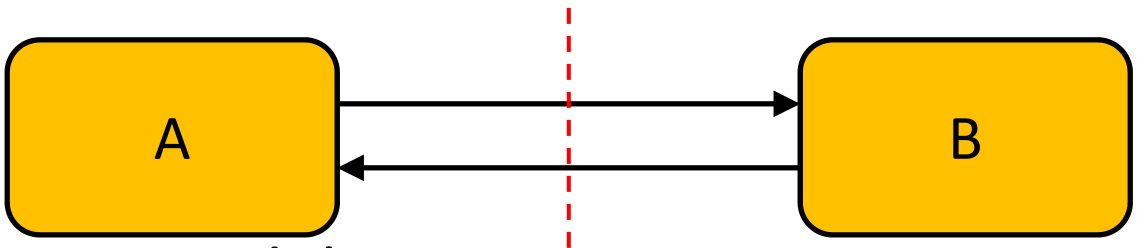
A set of views are consistent =

∃ **witness system** that could generate those views

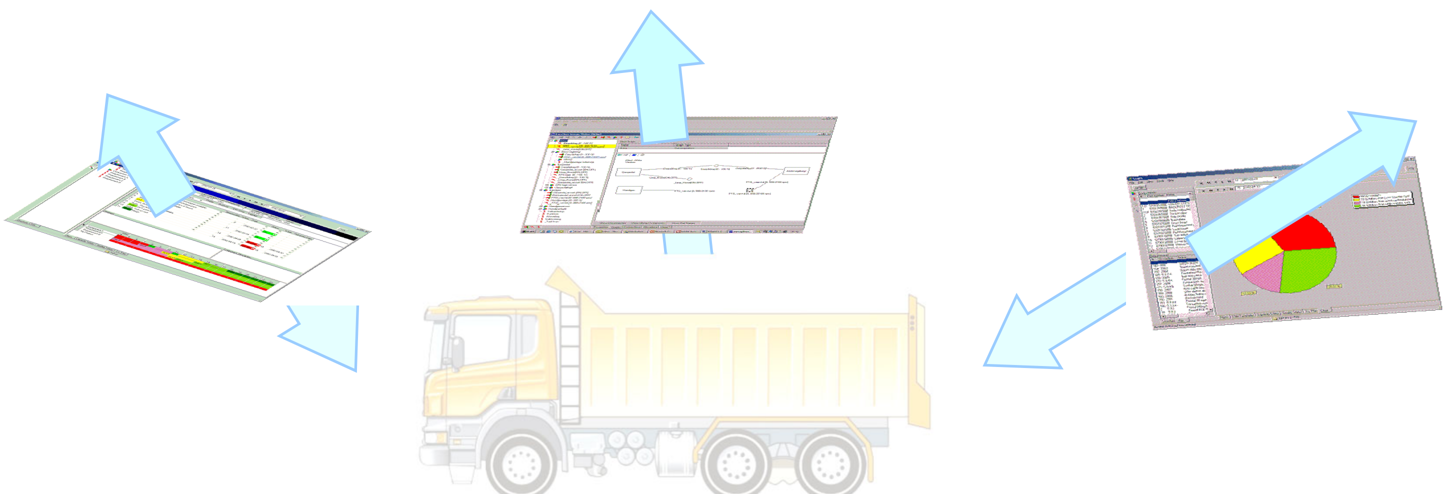


From (separate) components to (overlapping) views

- Refinement theories
  - model interacting but **separate** components



- Multi-view modeling
  - Views model **overlapping** aspects



Contributions

[TACAS'14, SAMOS'16, FACS'16, SoSyM'18]

- An abstract formal framework for reasoning about multi-view modeling
  - Systems and views are sets of behaviors, but generally in different domains.
  - Abstraction functions map system behaviors to view behaviors.
  - View consistency, synthesis, etc problems defined in this framework.

- Instantiation of the framework for various types of discrete systems and different types of abstraction functions:
  - Symbolic transition systems, finite regular and Buchi automata (regular and omega-regular languages).
  - Projections, periodic sampling, …

- Study decidability and complexity of checking consistency and synthesizing a witness system.

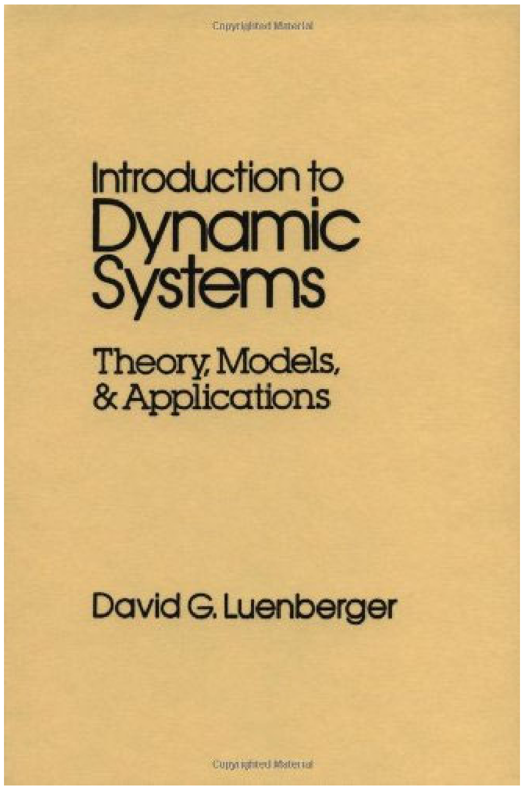
Stavros Tripakis

SOME THOUGHTS ON EDUCATION

What is the mathematics of the science of software?

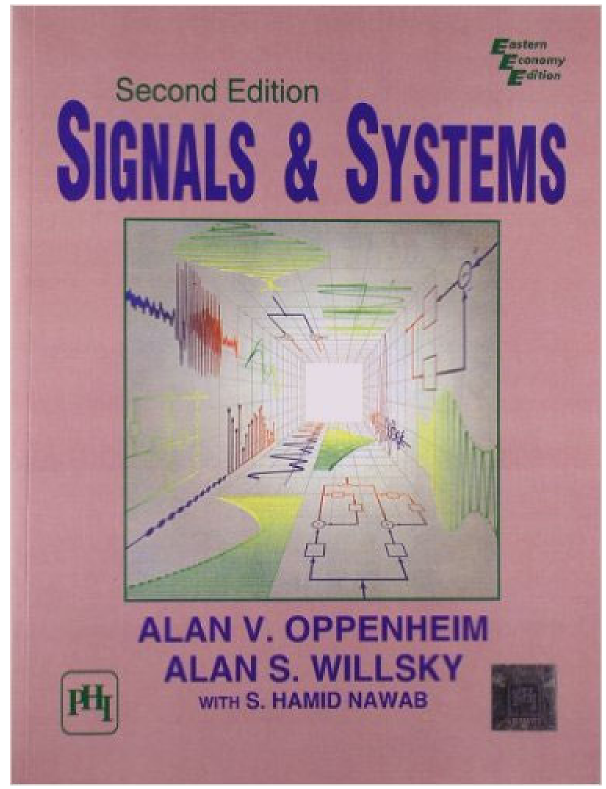
- Logic

Systems theory (classic)



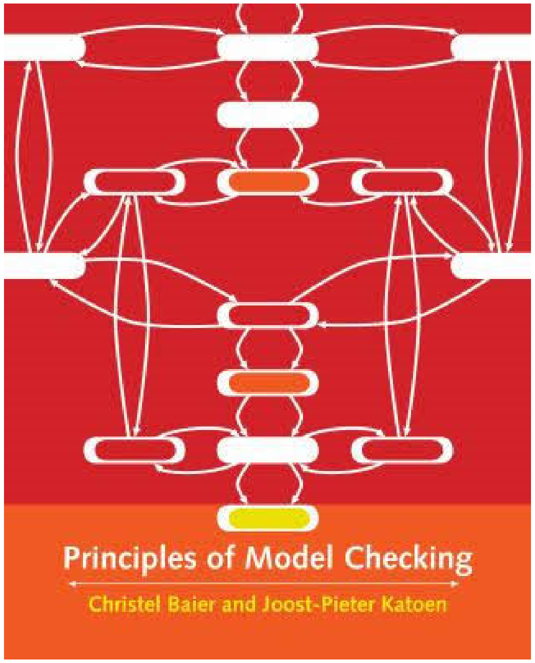
1979

1983

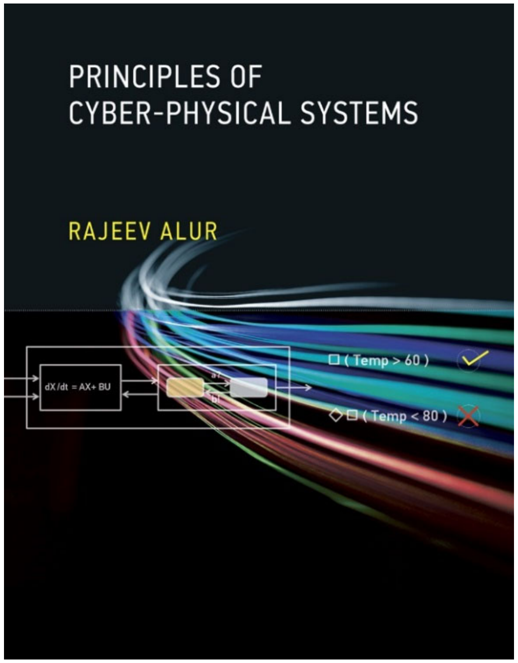


Study specific classes of systems (e.g., linear/non-linear differential equations)

Systems theory (modern)



2008



2015

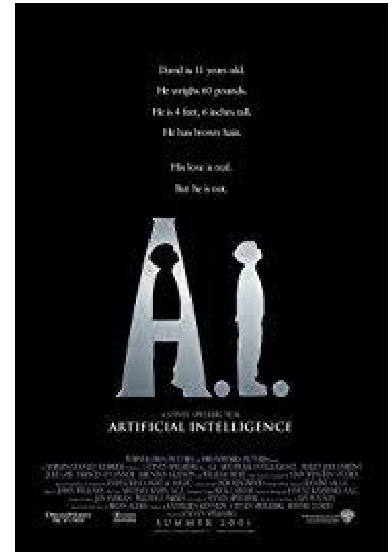
and many, many others …

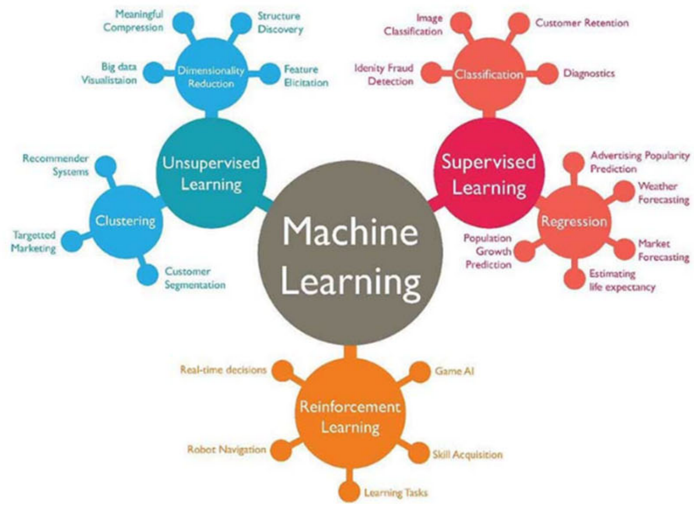
Study general and fundamental concepts to all systems (e.g., states, transitions, reachability, safety, liveness, fairness, correctness, refinement, compositionality, …)

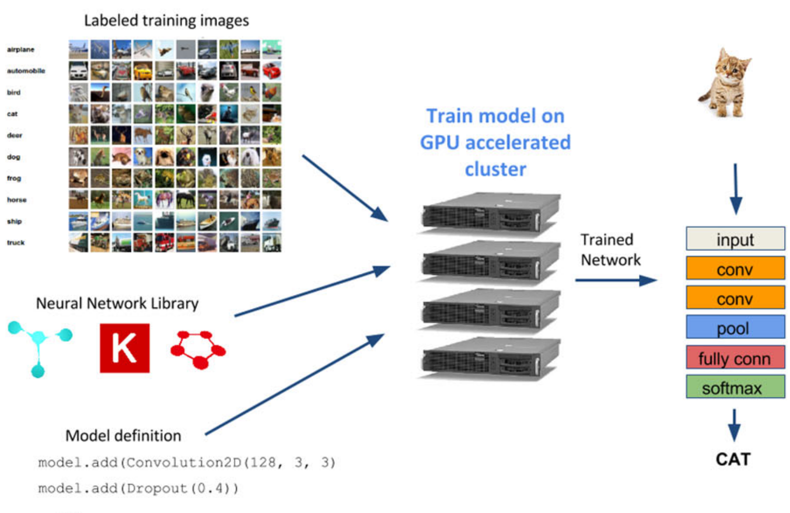
Stavros Tripakis

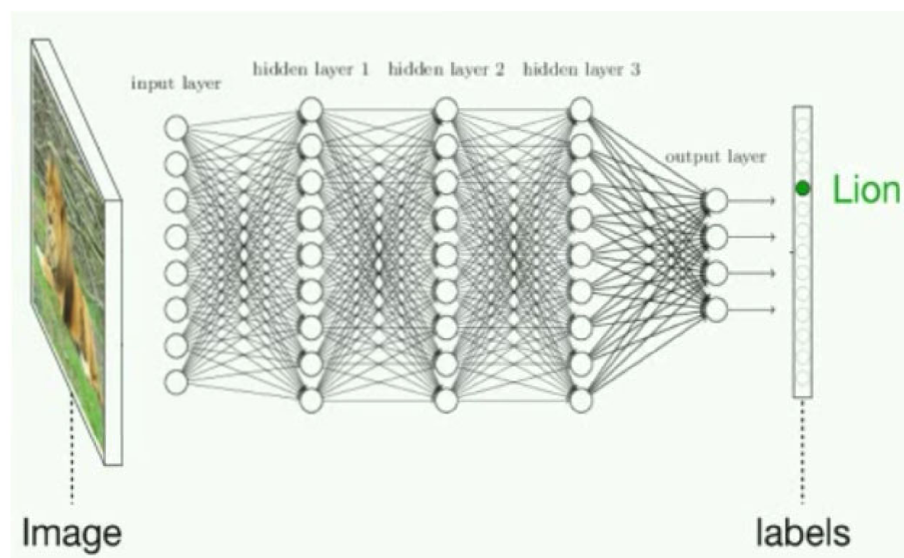
PERSPECTIVES

Brave new world









New challenges and opportunities

- Can AI benefit from the science of software, and how?

- Can the science of software benefit from AI, and how?

Can AI benefit from the science of software?

- 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 the science of software benefit from AI?

- Yes.

- Model learning (and its connections to machine learning).

- **Data-driven and Model-based Design (DMD)**
  - Top-down: from specification to implementation (specialization)
  - Bottom-up: learning from examples (generalization)

An example of DMD: 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

- We live in the world of software (and models of other systems, which are also software)
- Software is complex => difficult to get right
- Strong predictions about software require a hard science => formal methods, verification, synthesis
- AI/learning brings new challenges and opportunities

Thank you

- Questions?