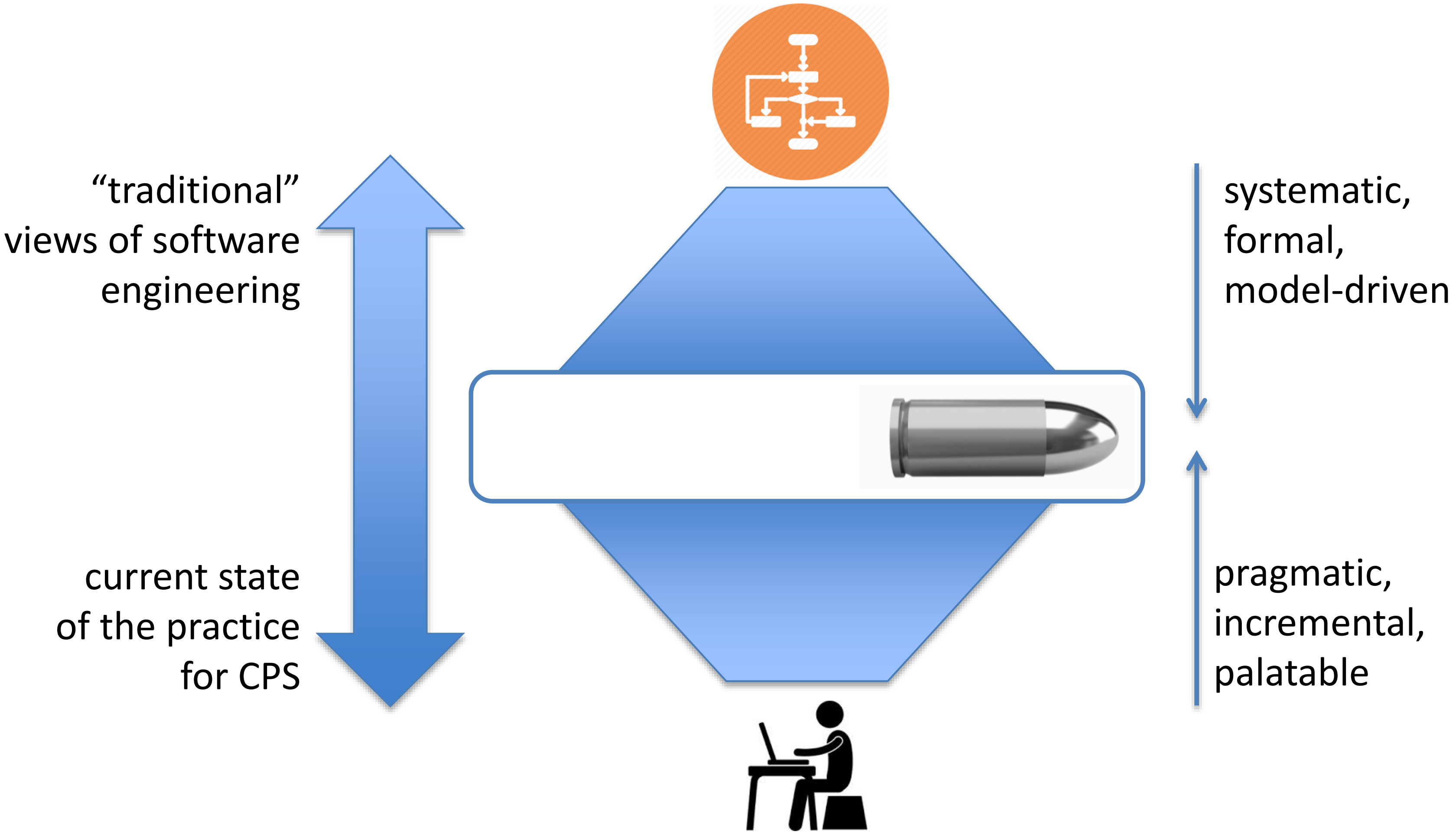
Pragmatic ~~vs.~~ and Systematic Engineering

Christian Berger
Stefan Biffl
Franck Fleurey
Scott Hissam
Christine Julien
Filip Krikava

Conceptual Summary



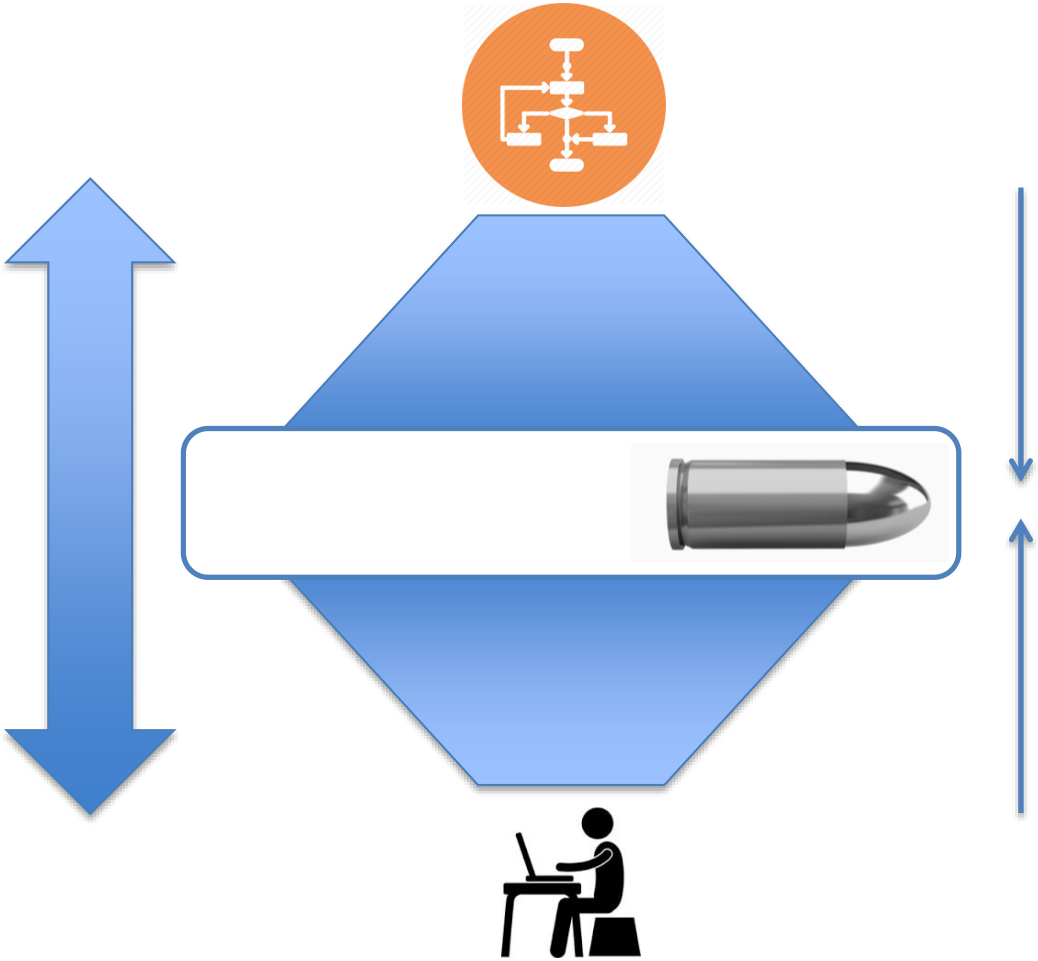
“traditional”
views of software
engineering

current state
of the practice
for CPS

systematic,
formal,
model-driven

pragmatic,
incremental,
palatable

Conceptual Summary



“mission critical” systems

“maker” systems, Internet of Things

systematic,
formal,
model-driven

pragmatic,
incremental,
palatable

The “Pure” Perspectives

- In “traditional” software engineering, we usually have the goal of abstraction
  - CPS may require balancing this goal with meeting domain experts “at their level”
- From a “purist” perspective, we need:
  - A clear, complete high-level description and a “compiler” that can generate correct code
  - Domain specific languages offer a first step
    - i.e., they make it possible to specify a solution and generate code automatically
  - But generated code is often “not efficient enough”

The Mismatch

- There is a key interdisciplinary problem
  - i.e., a “mismatch” in the languages spoken by the software engineers and the domain experts
- Often what happens is the domain experts build a system
  - When they’re desperate, they call in a software engineer to try to find the problem
  - The software engineer has to become immersed in the domain
- Recent anecdotal efforts hint at interdisciplinary teams that start with domain experts and software engineers

A Continuum of Systems

- There are different categories of systems
  - Mission critical ones may *require* a rigorous approach from the outset
  - However, more “user” level systems (e.g., in the IoT, “maker” systems, etc.) may not
- Systems are almost universally made up of components
  - There is a danger in the components being used for something unintended later

Some Directions

- Can we leverage system “smarts” to aid in high quality systems?
  - Maybe it’s ok for the system to have flaws, as long as it has the smarts necessary to recover or repair itself
- Can we derive models that are inherently composable?
  - We can then model components of a CPS that it is necessary to model and perhaps not others
  - Individual models may then be more tractable
  - “Connectors” across models could account for consistency, timing, security, etc.

Some More Directions

- We need to improve design-time techniques and make them accessible to domain experts
  - However, we may also want to do some of the checking at run-time using real world data
  - Design-time checking will necessarily be more open

A Caveat

- We are not going to replace the domain experts
  - The domain expert needs to be able to rely on tools to *introspect* expressively
    - Provable correctness (even at the component level)
    - Testing in the *your* environment (i.e., a well-defined test suite) – relationship to “certifiable” components
    - Smartness and self-adaptation
  - How do we make the domain experts trust the tools?
    - Time? Mixed teams? Technology transfer?
    - **Evidence**