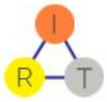
COLUMBIA UNIVERSITY

Making IoT safe: Expressing IoT safety, privacy and security policies

Luoyao Hao, Jan Janak & Henning Schulzrinne

Oct 2023



Internet Real Time Lab

SIP, NG-911, SECE, NetServ, DYSWIS, and Video streaming

“Safety is the avoidance of unacceptable hazards, including threats to human lives, the environment, or to costly facilities. Safety constraints are expressed using assertions that define system states that should not occur because they may lead to mishaps.” (Fernandez, 2017)

Reliability Metrics

Reliability: The probability that a system continuously delivers correct service up to time *t*, a.k.a. Mean Time Between Failures (MTBF)

Availability: A percentage of the time when a system is expected to be operational, a.k.a. “number of nines”

Serviceability: A mean time to repair failed system (a.k.a downtime)

Safety: the non-occurrence of catastrophic failures with consequences higher than the benefits provided by correct service

Fail-safe: Revert to a safe state in the event of a failure

Obtaining Dependability by Applying Constraints

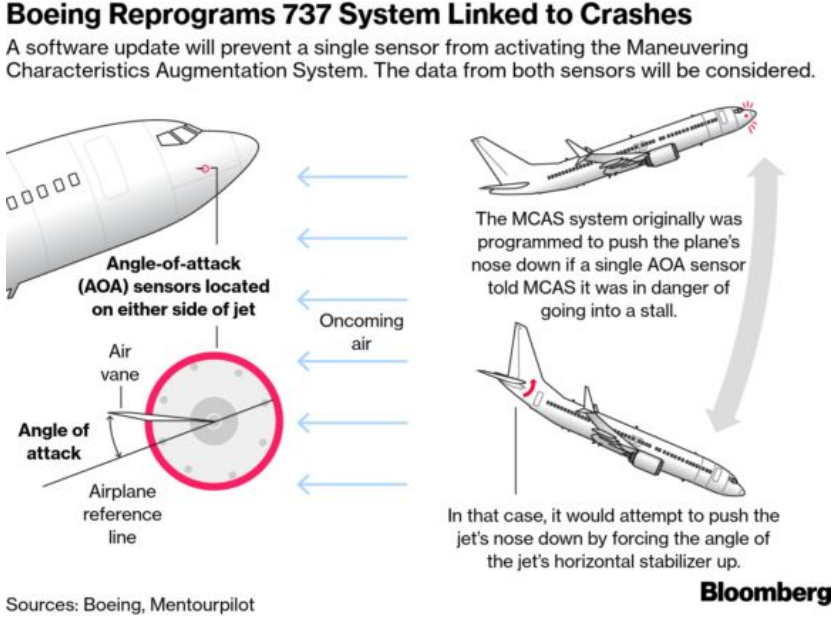
What to restrict:

- Who and when can access an IoT system
- Who can communicate with whom and when
- Type of computation performed by system components
- Amount of computation per system component
- Degree of freedom at application level

Dependability Research: The art of designing constraints
Dependability Engineering: The art of applying constraints

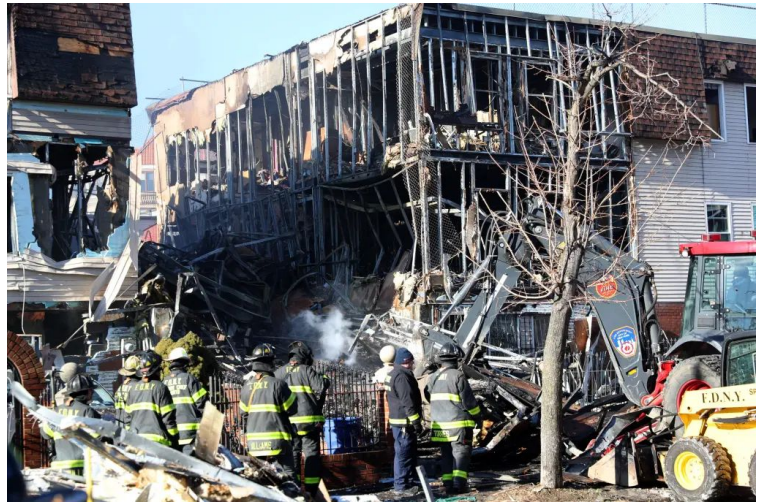
When systems fail to connect properly

- Avoidable tragedy or unavoidable bad connections



Boeing 737 Max was designed to react to only one of the plane's two "angle of attack" sensors that measure the jet's incline.

NYC ramps up inspections of self-closing apartment doors over a year after horrific Bronx fire that killed 17



The Bronx fire was blamed on a malfunctioning electric space heater and doors that didn't close properly.
"We'll continue to do everything in our power to make sure buildings are in compliance with our policies, including increasing enforcement under this new rule," - NYC Housing Preservation & Development (HPD)

Merrimack Valley Gas Explosions (2018)

- Sensor-actuator mismatch during system upgrade
- High-pressure gas released into a low-pressure system
- Monitoring center received alarms, but had no control over valves

Lessons Learned

- Managing system evolution is hard
- Lack of system description data (GIS)

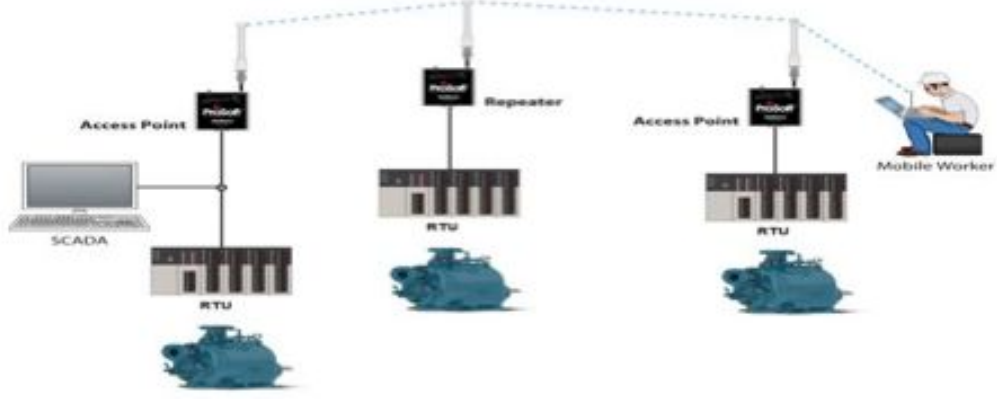
National Transportation Safety Board, "Overpressurization of Natural Gas Distribution System, Explosions, and Fires in Merrimack Valley, Massachusetts September 13, 2018", 2019, NTSB/PAR-19/02

Maroochy Sewage Spill (2000)

- SCADA pumps released sewage into river, local parks, and residential property
- Insider attack by a disgruntled employee in charge of upgrade
- *(DARPA concerned about this in the electrical grid)*

Lessons Learned

- Perimeter-defense won't stop insider attacks
- Need mechanism for temporary maintenance access
- Secure system logging is crucial



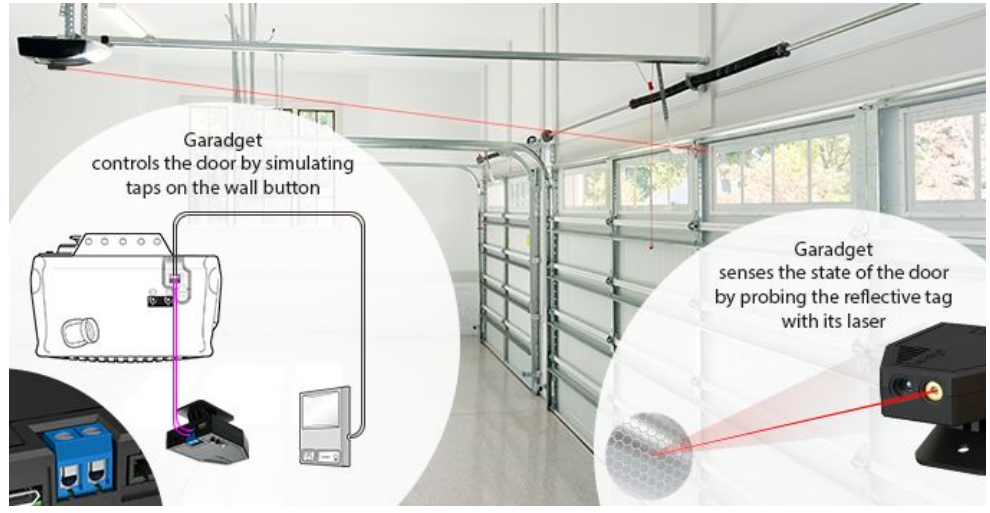
Slay, Miller, *Lessons Learned from the Maroochy Water Breach*, ICCIP 2007
Sayfayn, Madnick, "Cybersafety Analysis of the Maroochy Shire Sewage Spill", MIT, 2017, CISL# 2017-09

Garadget (2017)

Cloud-connected garage door controller

Bad Amazon review and angry language led manufacturer to brick user's device remotely.

<u>What can we do to prevent power abuse in cloud-based systems?</u>

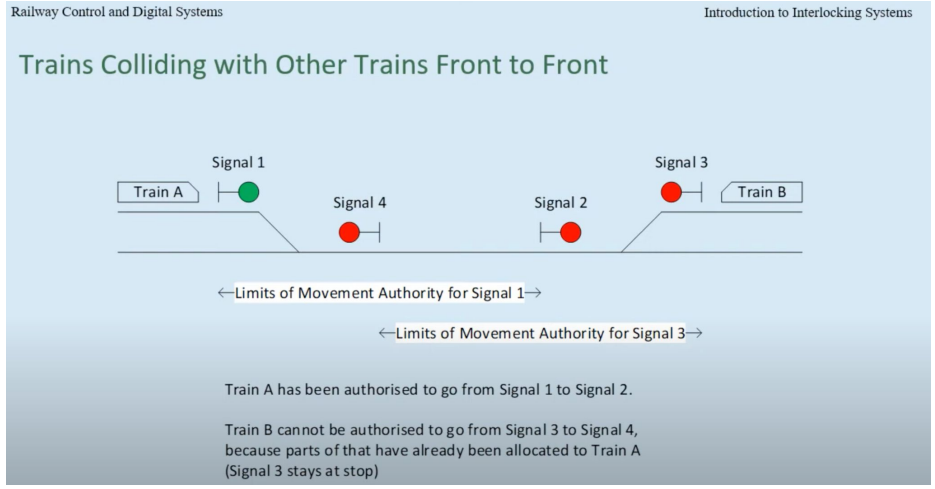


Gallagher, *IoT garage door opener maker bricks customer's product after bad review*, ArsTechnica, 2017

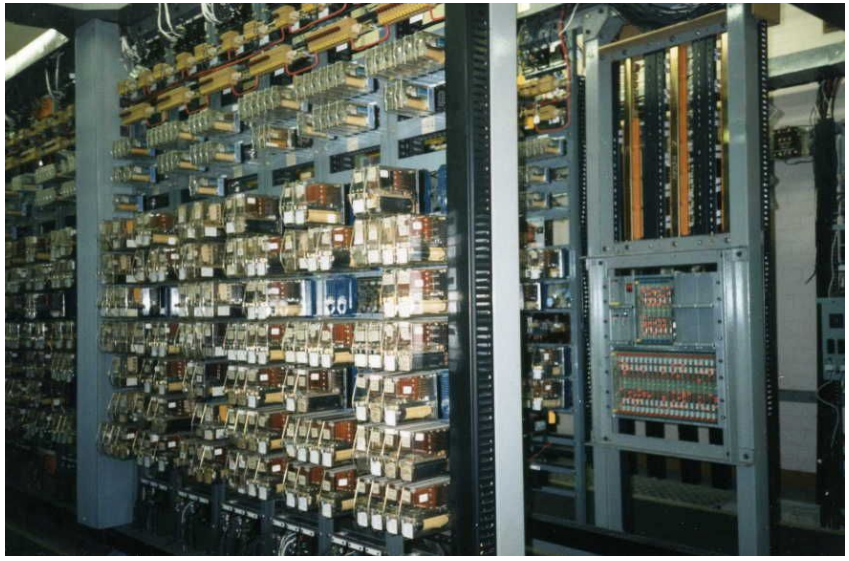
Classic example: Railroad interlocking

Railroads = sensors & actuators (switches & signals)

"An interlocking system is designed so that it is impossible to display a signal to proceed unless the route to be used is proven safe."



https://www.youtube.com/watch?v=Fqy4arxWmnQ



Safety and integrity rules

Classical computer system:

prevent unauthorized access -- integrity, availability

resource locking (semaphores, locks, transactions, …)

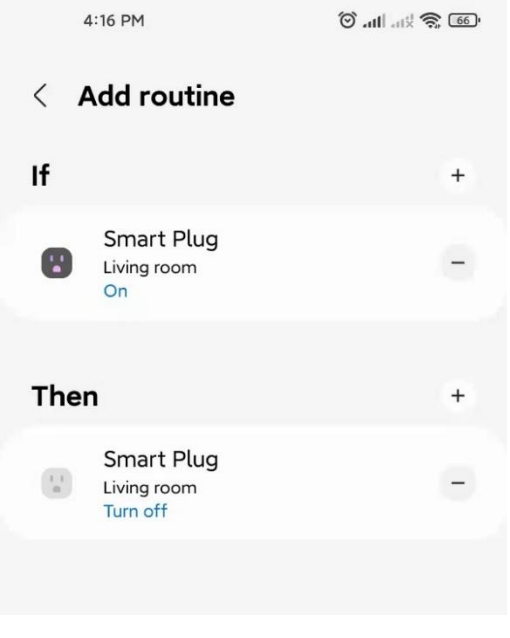
- *lockouts & interlocking*: “cannot start cleaning cycle if the door is open”
- *load threshold*: "the sum of power of devices must be < T"
- *timing constraints*: "Heat pump cannot be turned on if it was turned off less than 10 minutes ago"
- *actuator storms (*control instability or hunting*):* change in S activates actuator A, which changes S
- *safety limit constraints*: "Windows cannot be left open for more than 10 minutes when the outside temperature is below 40F"
- *resource constraints*: "Microwave ovens in kitchens cannot be turned on if we're in mandated ERLP Stage 1"
- *nuisance constraints*: "The trash compactors cannot be turned on between 9 pm and 6 am on weekdays and 9 pm and 10 am on weekends" (noise ordinances)
- *sanity constraints*: "Thermometer values must be between -20 and +40 degrees Celsius", "Relative humidity must not exceed 100%" (report an error otherwise)
- *liveness constraints*: "Must hear from the device no less than every X seconds", "warning when battery level drops below X"

All can be overridden by authorized users (e.g., building manager) or maybe several (two-person rule)

Connecting blindly can be a bad idea

- External dependencies
  - Amazon Echo accidentally shuts down itself by turning off a socket
- Internal dependencies
  - Smart plug deadlock IFTTT routine
    - “if on then off”
- Accidently injected by users/voice assistant (misheard) or intentionally leveraged by malicious devices
- Connection does not bring better interaction (or probably worse)
- Safety-critical, energy-critical command
  - “set room temperature lower than 50 °F”

Current IoT systems only have **single-device awareness**



Proving correctness & policy intervention

self-interest

liability

codes (e.g., electrical & boiler)

laws & regulation (FERC, DOT, FAA)

Prevent that any commands (automated or manual) cause a dangerous or undesirable state or action

Predict: Can the system do something dangerous or "stupid"?

e.g., "no dangling devices", "no direct connection between sensors and actuators of different types", "no out-of-range values for devices with a range"

Creating and maintaining policies is hard

- Policies shouldn't change if the hardware or network address is changed → naming
- Who will be editing the policy rules? Some administrator, or an end-user of the system?
- Start from scratch, or provide blueprints that you can import and perhaps customize?
- If there will be blueprints, how will they be distributed? What customization mechanisms does the blueprint provide so that it could be customized for a particular system/deployment?
- How does the end-user edit the policy?
  - Text file with the policy rules? Through a web-interface? In some other way?
- Is portability a concern?
  - Do you want to be able to create policies that are applicable to multiple deployments, e.g., multiple smart homes?
- Does the policy engine need to perform sanity checking when deploying a new policy, i.e., at compile-time, for example, to detect conflicts?

Trade-offs for languages

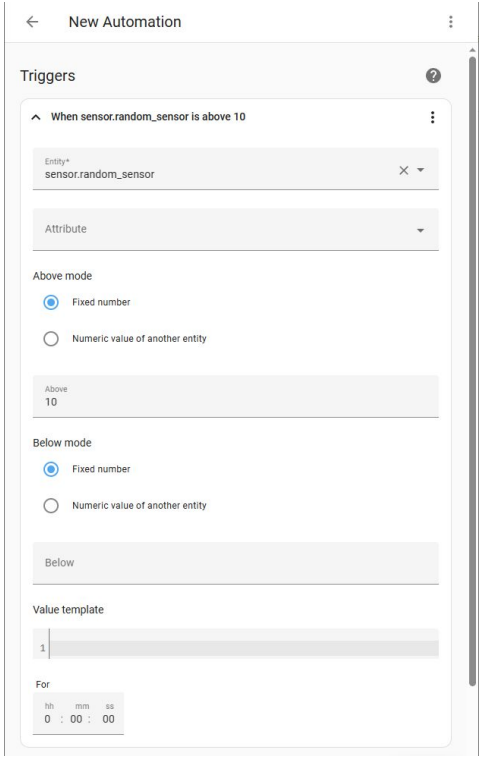
- *Programming paradigm*:
  - imperative, procedural, object-oriented, declarative, functional, reactive, …
  - e.g., Python vs. YAML
  - familiar to
- *Brevity*: The shorter, the better.
- *User interface:* cannot build a web interface for a general language, but can for YAML
- *Maintainability*
  - Does the language provide support for comments?
  - Can you split the policy into files or modules? Can policy versioning be supported?
- *Extensibility*:
  - Encapsulation or abstraction, i.e., can you build more complex rules out of simpler ones without having to know about the implementation details of the building blocks?
  - Inheritance or specialization, i.e., the possibility derive new rules from existing ones
- *Specificity*: Domain-specific (e.g., HA) or general, i.e., capable of expressing policies from a variety of application domains?
- *Compile-type checks*: e.g., static typing?

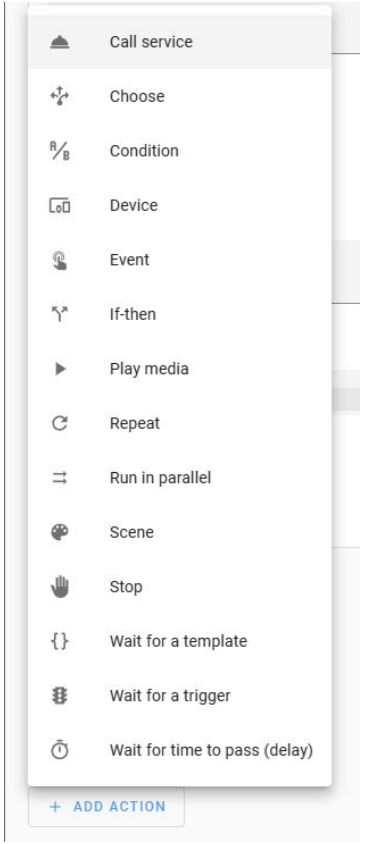
Example: Home Assistant = trigger, condition, action

(trigger) When Paulus arrives home

(condition [= state]) and it is after sunset:

(action) Turn the lights on in the living room

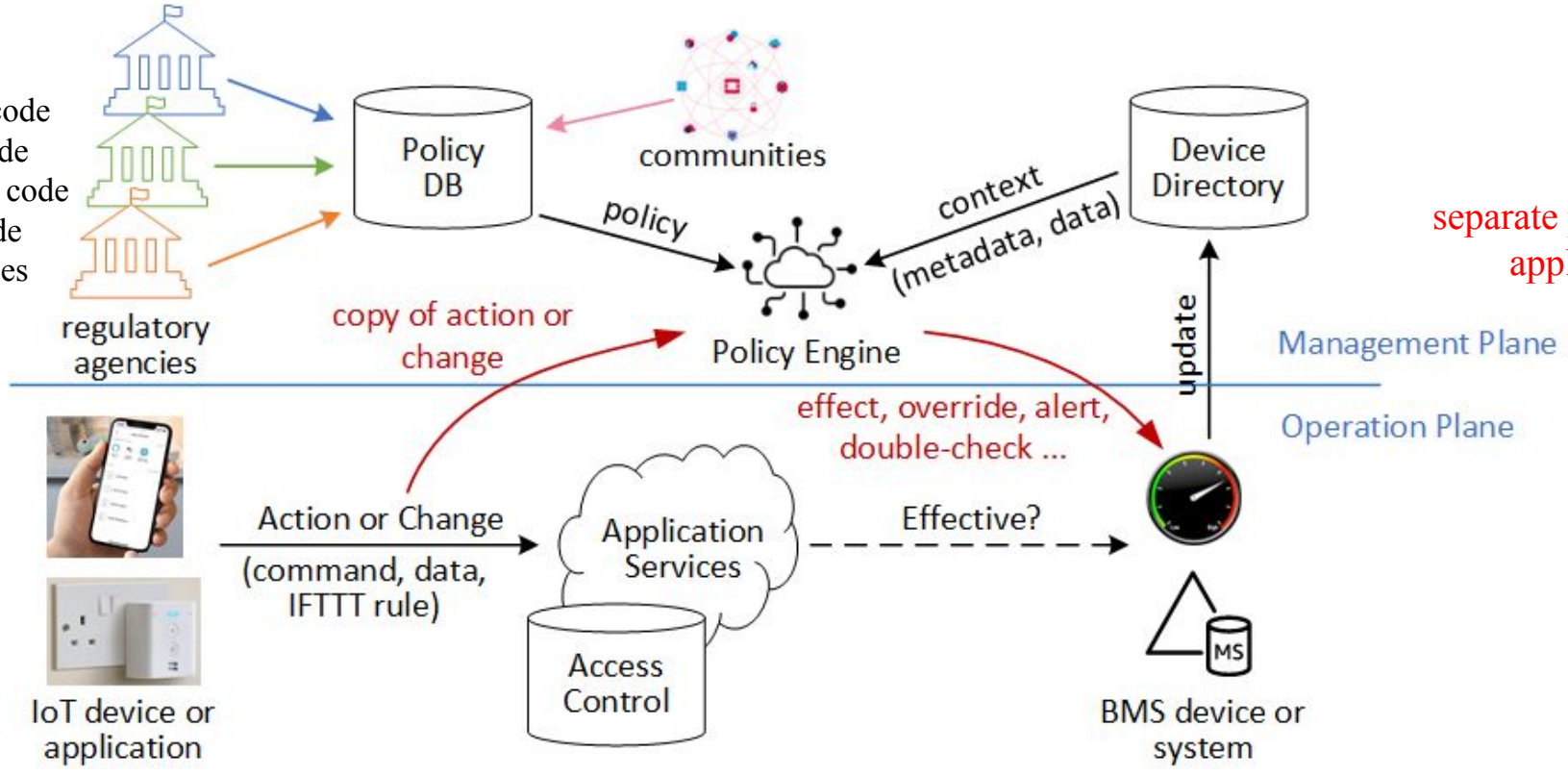




The “Management Plane” for IoT systems is missing

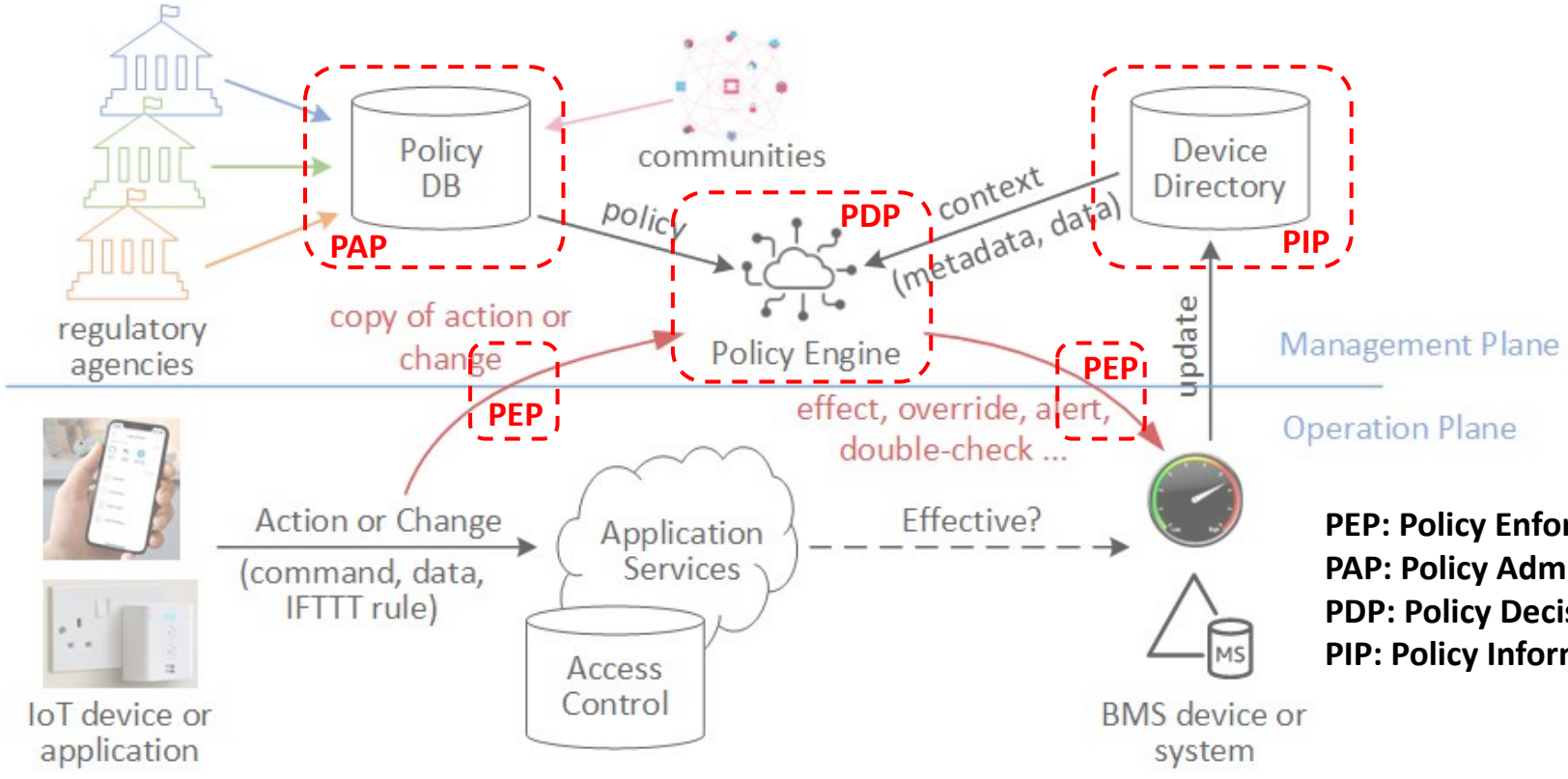
– a management plane to prevent systems from doing stupid or undesired things

fire code
electricity code
building code
mechanical code
fuel gas code
regional rules
…



separate policies from applications

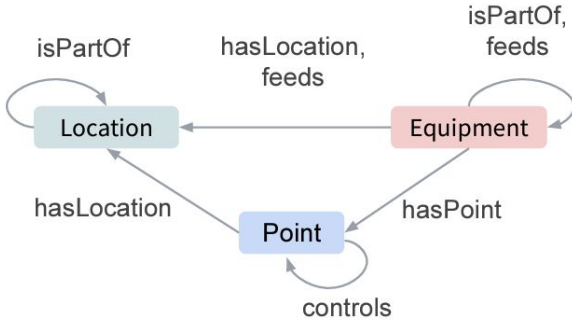
Analogy to authorization service implementation



PEP: Policy Enforcement Point
PAP: Policy Administration Point
PDP: Policy Decision Point
PIP: Policy Information Point

Recap: what's there

- Some ontologies to describe relationships
  - E.g., Haystack, IFC, Brick
  - Location, service relationships



Modeling Support	Brick	Project Haystack	IFC	BOT	SAREF
HVAC Systems	**yes**	**yes**	**yes**	no	no
Lighting Systems	**yes**	partial	**yes**	no	no
Electrical Systems	**yes**	**yes**	**yes**	no	no
Spatial Information	**yes**	no	**yes**	**yes**	no
Sensor Systems	**yes**	**yes**	generic	no	**yes**
Control Relationships	**yes**	no	generic	no	no
Operational Relationships	**yes**	no	generic	no	no
Formal Definitions	**yes**	no	**yes**	**yes**	**yes**

Some takeaways

- Relationships are well-defined for smart building
  - Wide representativeness, detailed classes and hierarchy
  - Interoperability, heterogeneity, visualization
  - Location (connected to, pat of) and function (input, output, serve)
- Not much room to improve them, unless:
  - Include user-to-device relationship (ownership)
  - Make clear difference between smart building and smart home
- Unclear how to make them useful for a large system
  - Defined but not used

Given well-described devices and relationships, what can we do to improve IoT management systems?

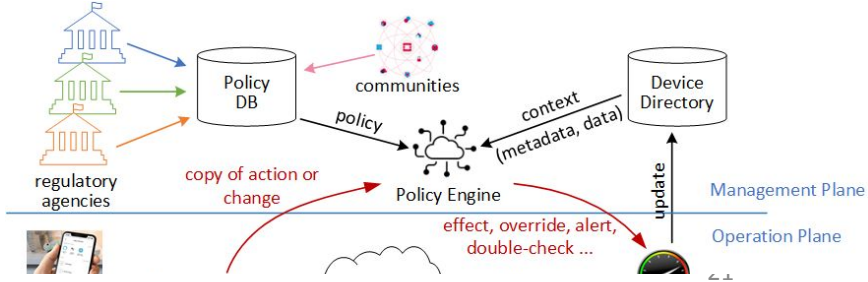
Comparison - Languages

	Brevity (p)	Brevity (e)	Type checking	Expressiveness	Ease of use	Implementation	Portability
Rego			dynamically	6 of 6			
Casbin			dynamically	2 of 6			
Sentinel			statically	6 of 6			
Polar			statically	2 of 6			
Python			dynamically and statically	6 of 6			

Expressiveness: how many examples can be implemented by the languages without further coding in the engine
Implementation: libraries are available for imports and other languages
Portability: ease of sharing policies

The “Management Plane” for IoT systems is missing

- Policy specification and policy language
  - Define backward compatible policies for huge number of things
    - properties + relationships, rather than identities
  - Authorization-based policy languages are ill-suited
    - only define can or can’t do
    - have to provide context beforehand
    - we may need a more generalized language (e.g., regular programming language)
- Standard interfaces: east-west, north-south
  - Retrieve policy, retrieve context from directory
  - Intervene devices or systems
  - Trigger actions on external devices



Policy specification: minimum needs

```
solar_power_policy = Policy(
    policy_id="policy_4",
    description="Deny: power on a device, when the rated power of the device"
                "plus the total power of the existing devices powered by the"
                "same solar panel exceed the limit power of the solar panel.",
    matching_attributes=MatchingAttributes(
        policy_subject=["type", "properties.rated_power"],
        policy_object=["type", "properties.limit_power", "relationship.powered_by"]
    ),
    policy_type="S2",
    policy_subject={"type": "device"},
    action="power_on",
    policy_object={"type": "solar_panel"},
    relation="policy_subject.relationships.powered_by.id == policy_object.id",
    assertion=power_limit_assertion,
    response="Deny",
    alert="Powering on this device will exceed the solar panel's power limit."
)
```

annotation
(not for process)

refer to metadata standard

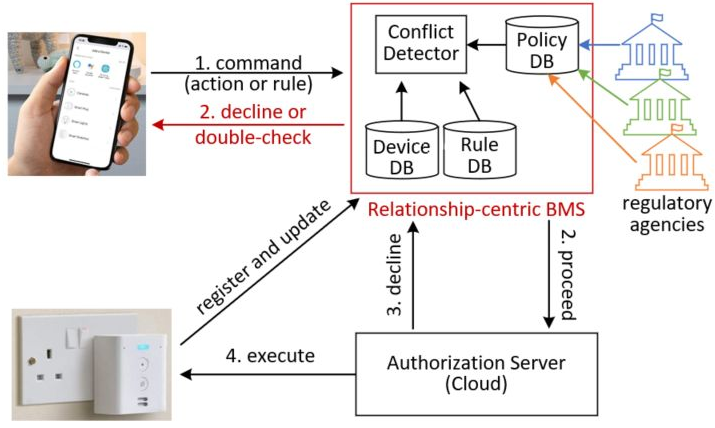
type of policy - to classify different parsers of policies

properties of subject, object, and their relationships and the action - to match policy from input and to find impacted devices from DB

assertion to evaluate, can be a function

result and instruction to the operation plane

System model



- Evaluate an incoming command against policies
- Command: an action or a rule
  - e.g., turn on the light; if the door is open, turn on the light
- Distinguish “rule” and “policy”
  - Rule: if this then do that ⇒ what should happen
    - triggered by events or user command (“Alexa, good night”)
  - Policy: safety or energy critical, or common sense ⇒ what should **not** happen
- Policy server is relationship-based
  - No device identity in policy

Design principles for the policy plane

- What's popular and usable should remain unchanged
  - Rule-based behavior, easily changeable
  - Reuse existing ontologies as much as possible
- Allow multiple types of participants
  - Clean separation of their obligations
  - Inspection before installation and replacement
  - Backward compatibility policy
  - Smallest and reusable set of policies
- Clean separation between policy plane and operation plane
  - Operation plane should keep intact (UX unchanged)
- Interference and Interaction
  - More double-check than deny policy, as situations can be complicated
  - e.g., fire + water leak: turn on or off sprinklers

These are why we must use relationships and why we must define a new policy specification

Summary: we need a policy plane

- Management Plane: policy-centric logics to bridge “isolated islands”
  - Policies, metadata, relationships
  - Unbind policies and identities for backward compatibility and reusability
  - Brings regulatory participants to large IoT systems
- Unprecedented automated and programmable IoT systems
  - Mitigate risks of harmful commands, wrong data, or malicious devices
  - Highly controllable, integrated system
    - e.g., Limit the power consumption lower than 20 kWh per day