17-423/723: Designing Large-scale Software Systems

Automated Design Analysis

April 9, 2025

Learning Goals

- Describe the limitations of testing for ensuring design quality
- Describe formal methods as an alternative approach to analyzing software designs
- Describe two types of formal methods: Model checking and automated reasoning
- Describe the potential benefits and limitations of formal methods

Software Quality Assurance & Testing

“We have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing.

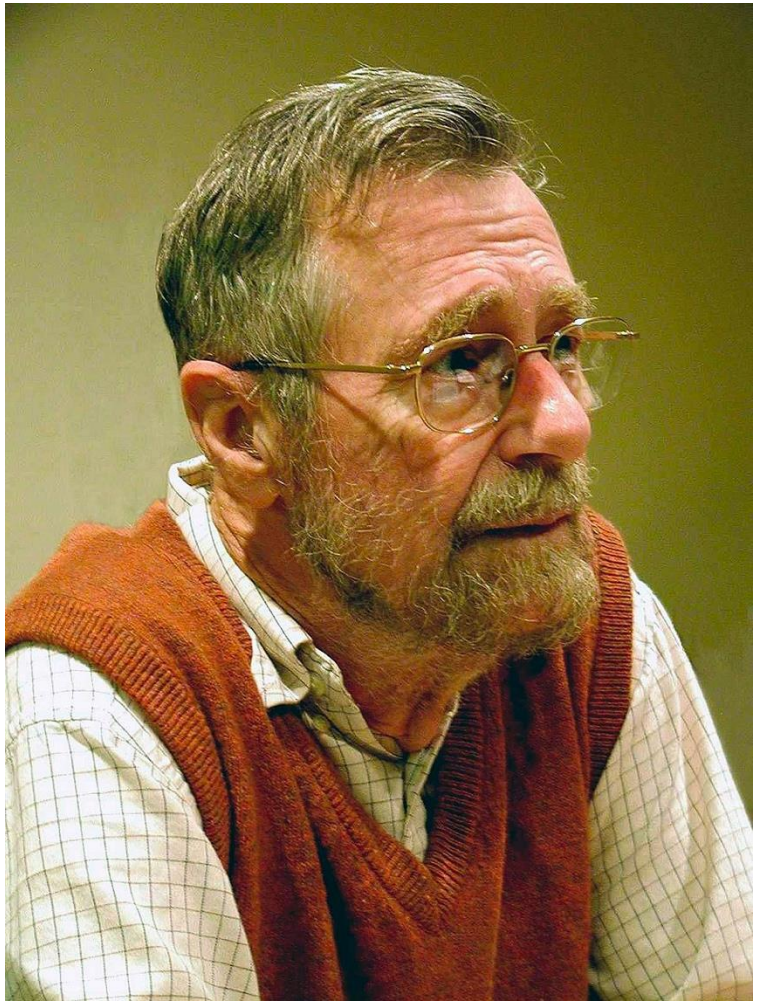
We're more of a testing, a quality software organization than we're a software organization.”

- Bill Gates

Testing: Limitations?

Q. Based on your experience, what are some challenges and limitations with testing?

Testing: Limitations?



“Testing shows the presence, not the absence of bugs.”

- Edsger W. Dijkstra

Formal Methods

- A class of techniques for ensuring software quality
- **Goal**: Provide strong, **mathematical** guarantees about the properties or behavior of software
- Types of formal methods:
  - Model checking
  - Automated reasoning
  - Interactive theorem proving
- Different methods, different levels of automation and guarantees provided
- A wide range of applications: Security analysis, bug finding, configuration analysis, program synthesis, etc.,

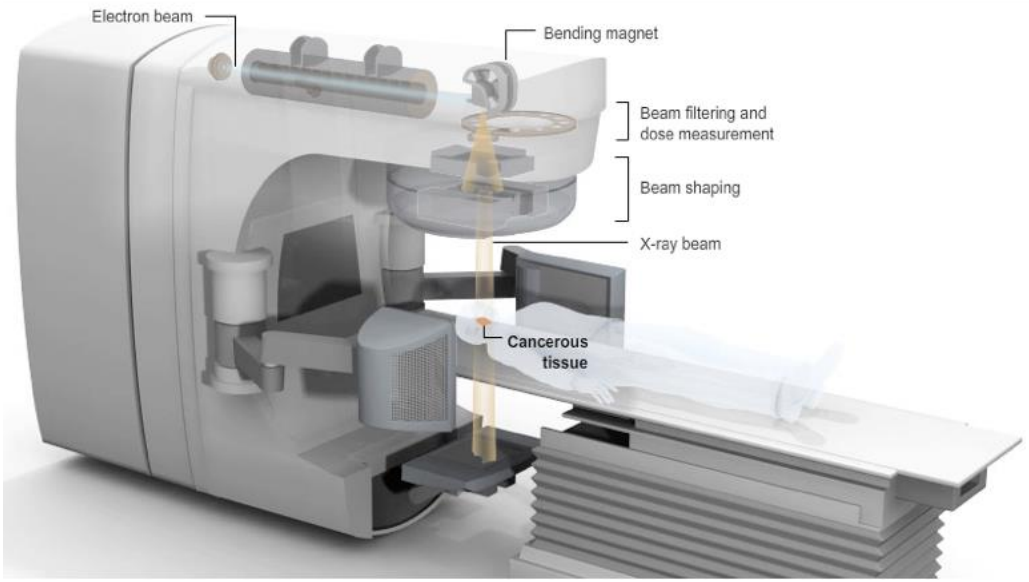
Formal Methods

- A class of techniques for ensuring software quality
- **Goal**: Provide strong, **mathematical** guarantees about the properties or behavior of software
- Types of formal methods:
  - **Model checking**
  - Automated reasoning
  - Interactive theorem proving
- Different methods, different levels of automation and guarantees provided
- A wide range of applications: Security analysis, bug finding, configuration analysis, program synthesis, etc.,

Example Domain: Medical Devices

Radiation Offers New Cures, and Ways to Do Harm

By WALT BOGDANICH JAN. 23, 2010

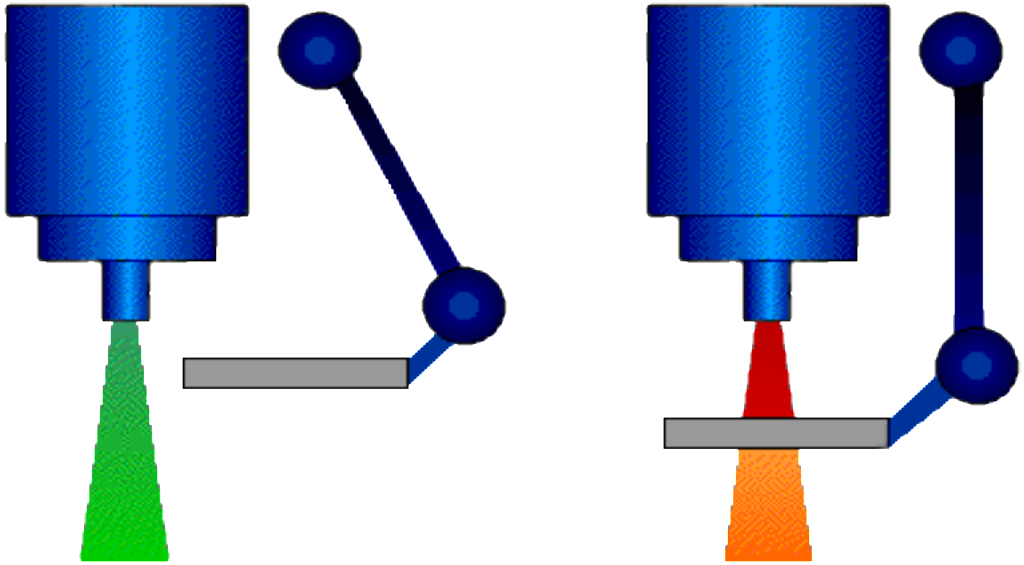


- 80,000 deaths & 1,700,000 injuries from medical devices since 2008 (in US)
- Actual numbers & root causes unclear; often unreported

Therac-25 Radiation Therapy Machine

- Used to treat patients at multiple hospitals in US & Canada during 1980s
- One of the most infamous examples of software failures in history

Therac-25: Radiation Modes



Electron Mode

Low power

Wide area

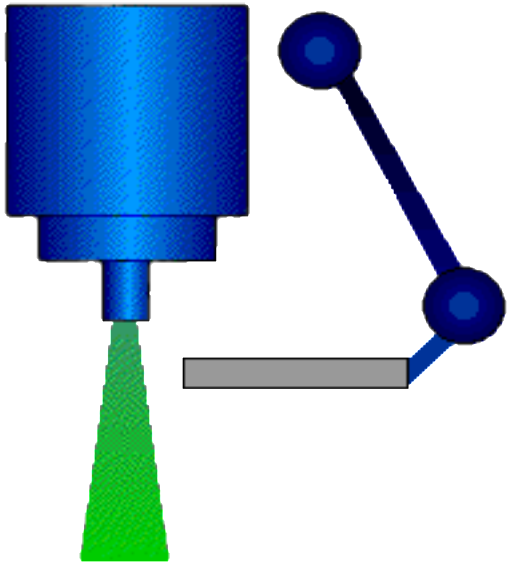
X-Ray Mode

High power

Narrow area

- Two different modes of operation: Electron-beam (Ebeam) and X-ray modes
  - A therapist enters the patient information and mode of treatment
  - In X-ray mode, a shield is inserted into the beam path to reduce treatment area

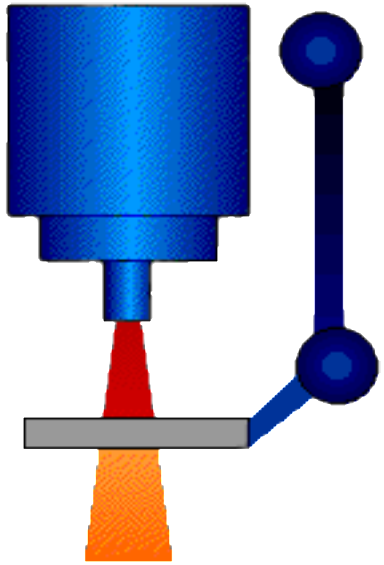
Therac-25: Radiation Modes



Electron Mode

Low power

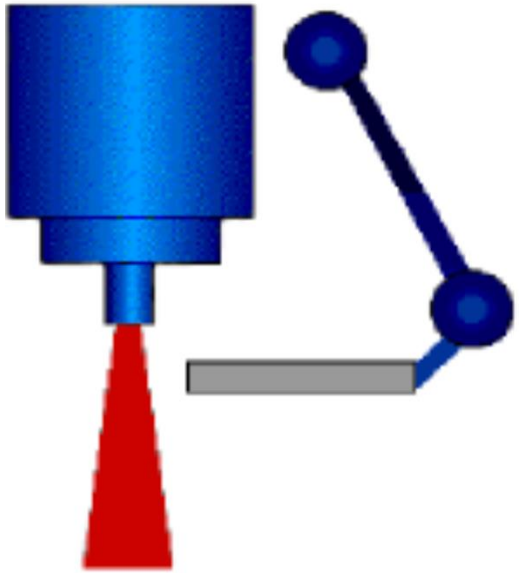
Wide area



X-Ray Mode

High power

Narrow area



THE PROBLEM

High power

Wide area

- **Failure**: In certain situations, the shield was not in place during the X-ray mode
  - Caused radiation overdose in some patients by up to 100 times
  - Killed/seriously injured 6 patients

Therac-25: What happened?

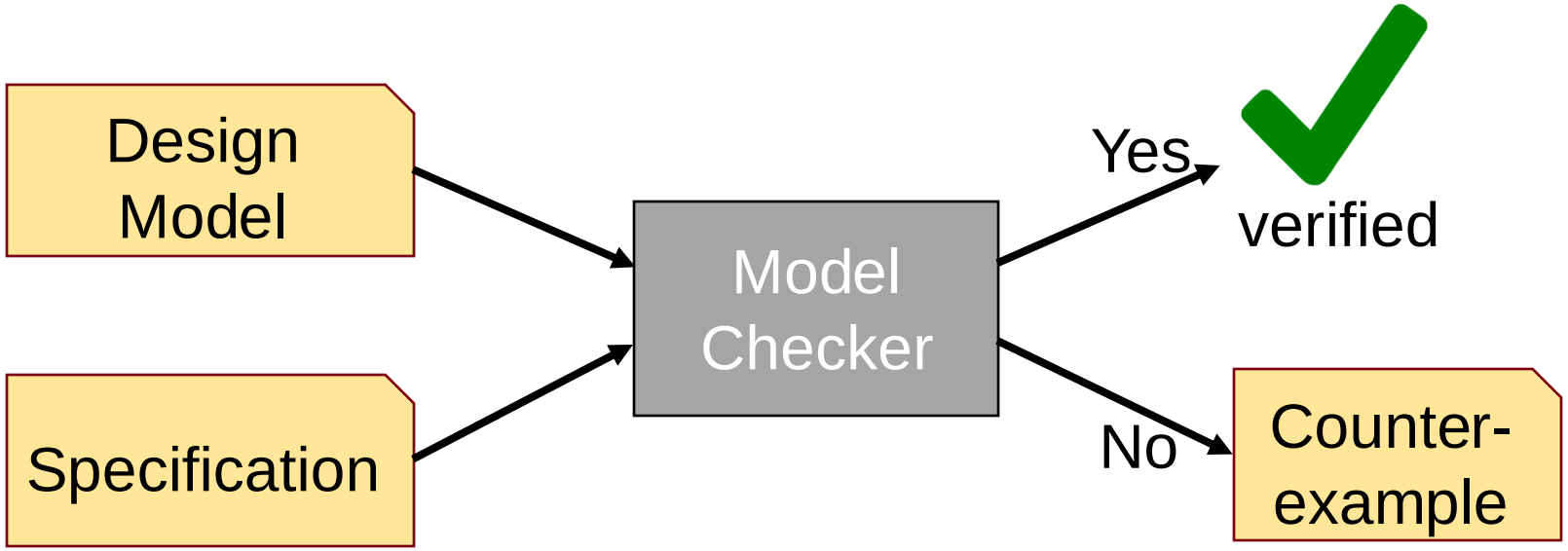
- Lack of robustness in software design
  - Against (easily predictable) human errors
- Reckless reuse of code
- Lack of proper software engineering practices
- General lack of concerns for software safety

Operator Error in Therac-25

“…[Therapist] noticed that for mode she had typed "x" (for X ray) when she had intended "e" (for electron)…the mistake was easy to fix; she merely used the cursor up key to edit the mode entry.”

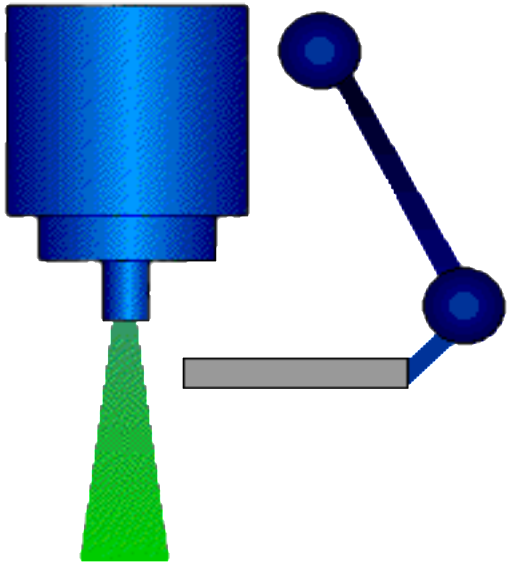
An Investigation of the Therac-25 Accidents
Leveson & Turner, IEEE Computer, 1993

Model Checking



- An approach for automatically checking software design to find errors
- Input
  - **Design model**: A formal, mathematical model of a system design
  - **Specification**: A formal statement of what it means for the design to be “correct”
- Output
  - A **counterexample** that demonstrates how the system fails its specification

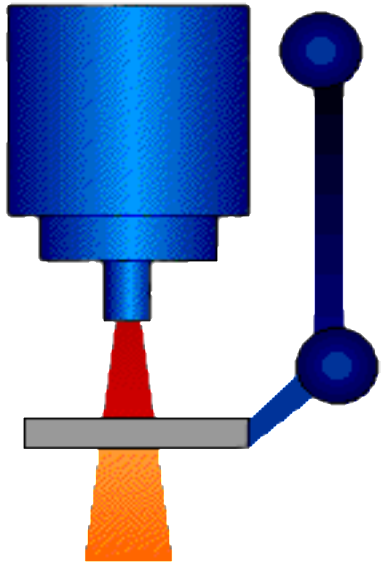
Back to Therac-25



Electron Mode

Low power

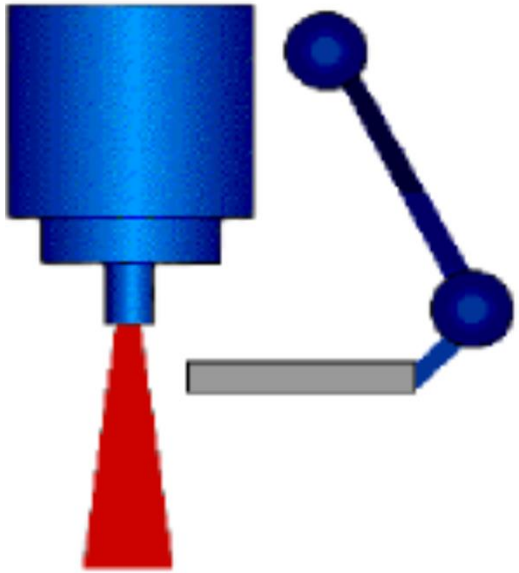
Wide area



X-Ray Mode

High power

Narrow area



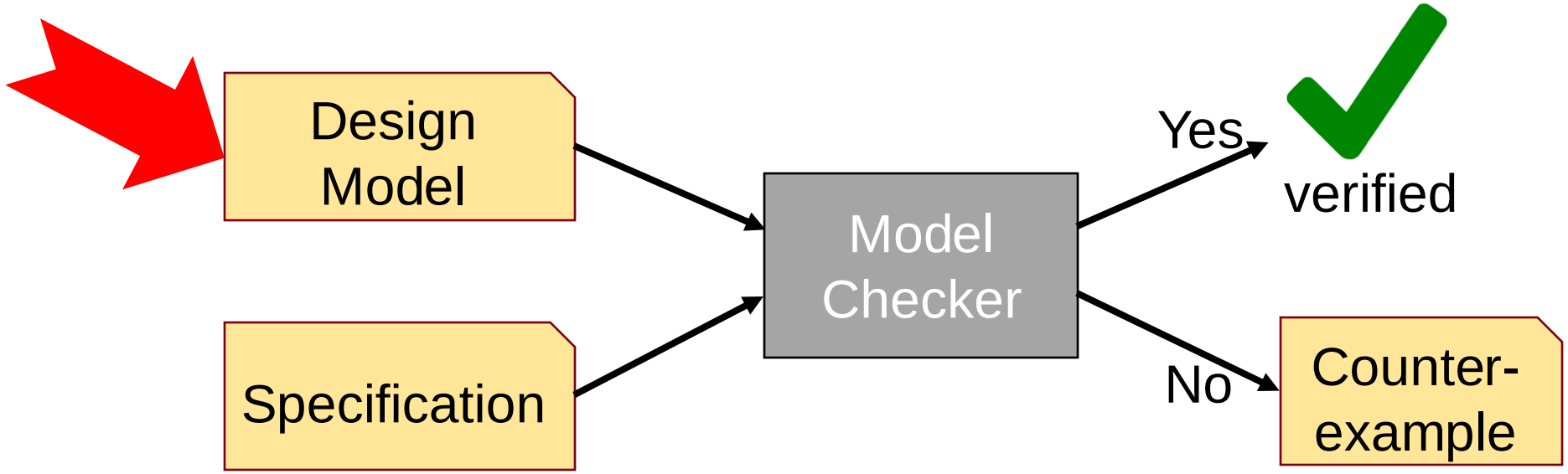
THE PROBLEM

High power

Wide area

- The failure was in part caused by a **race condition** between the user interface and the beam controller
- Let's see how model checking could have been used to find this problem

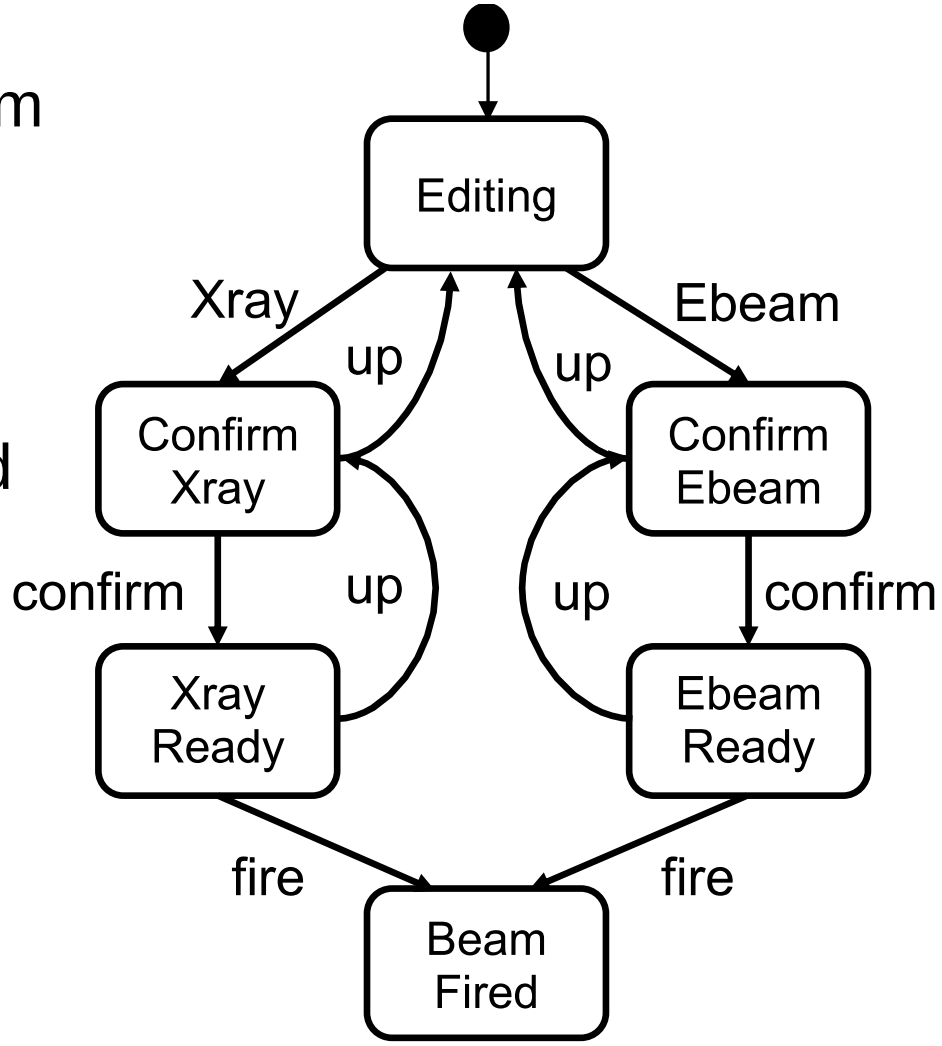
Design as State Machines



- **Design model**: A formal, mathematical model of a system design
- Another common type of design model is called **state machines**
- Each state machine consists of **states** and **actions**
- A machine moves from one state to another state by performing an action

State Machine: Examples

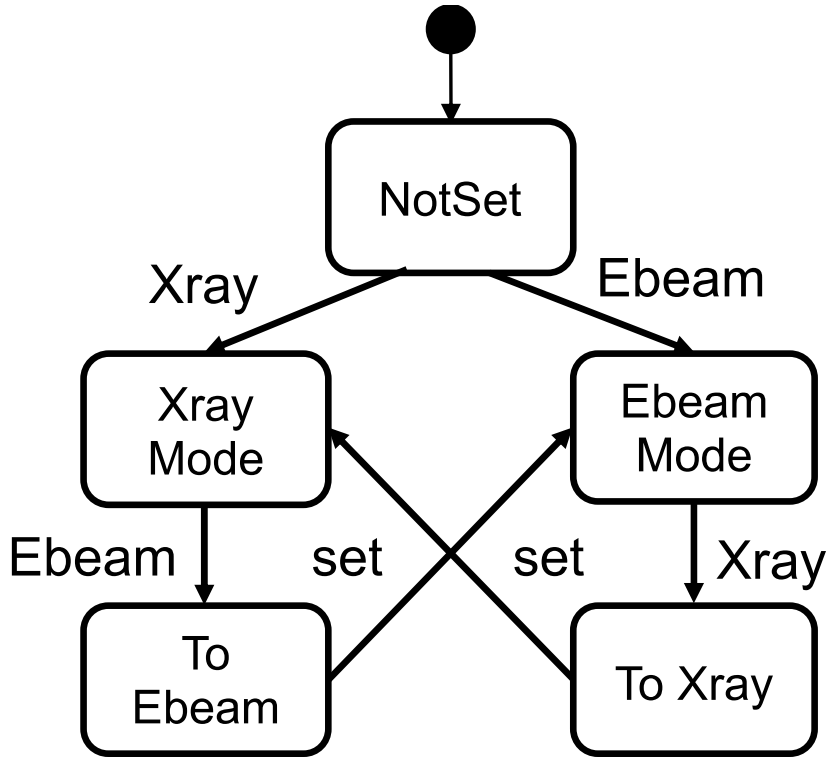
- State machine for the UI of the Therac-25 system
- Typical user task:
  1. Select X-ray or electron beam
  2. Confirm mode
  3. Fire the beam
- An execution of the system (a **trace**) is captured as a sequence states
- There are many possible traces
  - <Editing, ConfirmXray, XrayReady, BeamFired>
  - <Editing, ConfirmEbeam, EbeamReady, BeamFired>
  - <Editing, ConfirmXray, Up, Editing, ConfirmEbeam, …>
  - **Q. How many traces are there?**



Interface M_I

State Machine: Examples

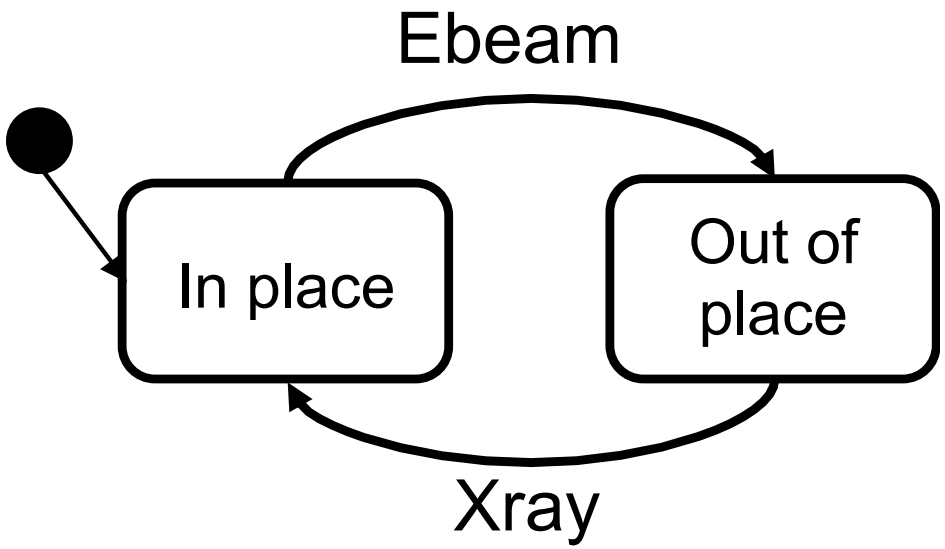
- A typical system will contain multiple, separate state machines (for different components)
- This is a state machine for the beam mode setter
- When Xray/Ebeam command is sent from the UI, it changes the mode of radiation
- There's a small delay when switching between the two modes (ToEbeam and ToXray states)



Mode Setter M_B

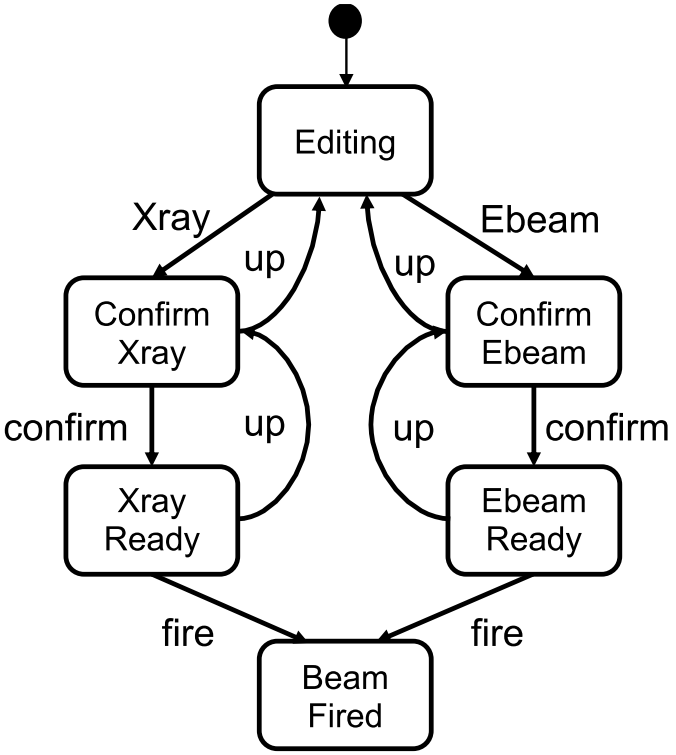
State Machine: Examples

- State machine for the shield
- The shield is inserted when the user selects the X-ray mode
- The shield is taken out when the user requests E-beam mode

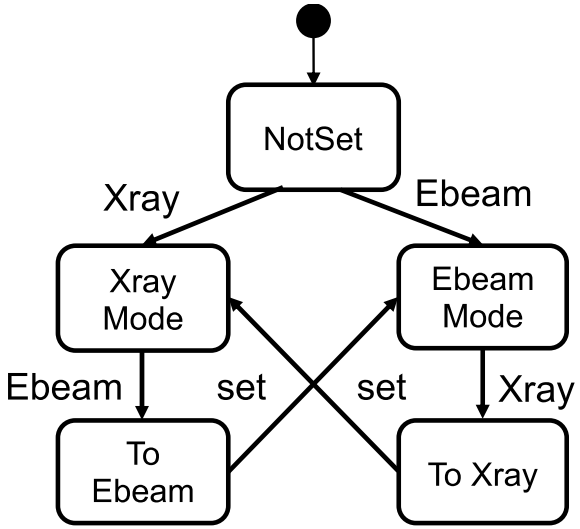


Spreader M_S

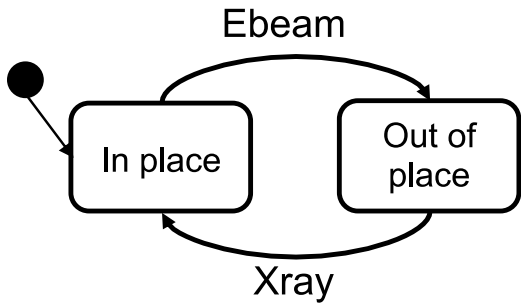
Overall Design Model for Therac-25



Interface M_I



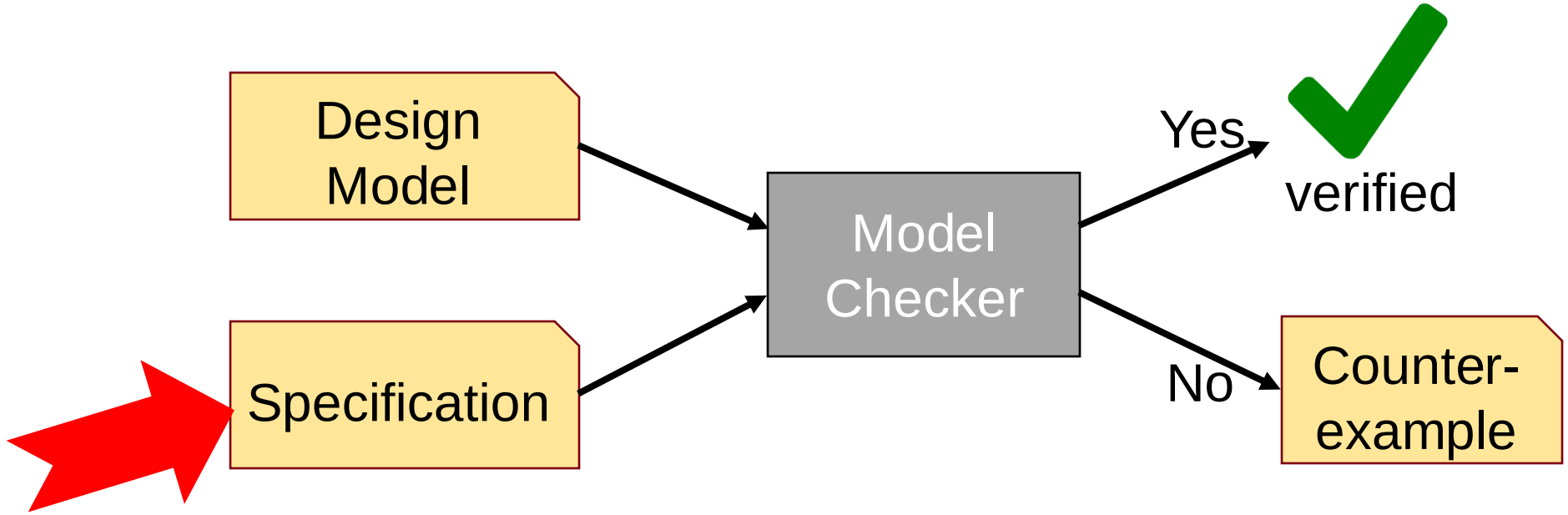
Mode Setter M_B



Spreader M_S

- The overall design model is a **composition** of the three state machines
- The state machines interact by performing common actions together & making state transitions **in parallel**

Specification



- **Specification**: A formal statement of what it means for the design to be “correct”
- There are many different languages for writing specifications
- We will talk about one type of specification language called **temporal logic**

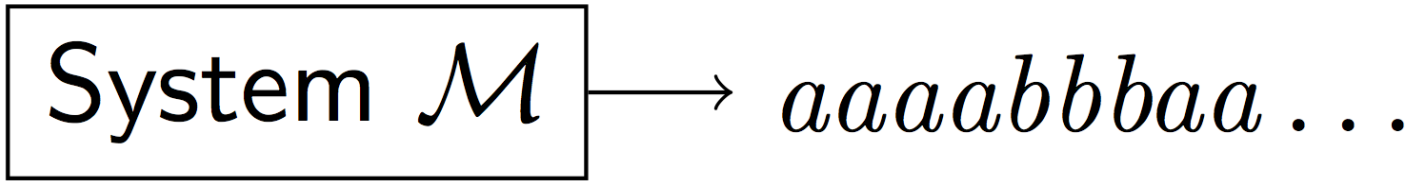
Temporal Logic

- A type of formal language for specifying behaviors of a system
- Describes desired behaviors of a system over time
- **Examples**:
  - Robot must eventually reach its destination
  - Vehicle should always maintain a safe distance to other cars
  - Robot waits until the obstacle is removed from its path

The Temporal Logic of Programs
Symposium on Foundations of Computer Science (1977)
ACM Turing Award, 1996

Amir Pnueli

System Behavior as Traces



- **System behavior:** An infinite sequence (or trace) of observations
- **Observation**
  - A state (e.g. “Robot is in shutdown mode”)
  - An action (e.g. “Robot moves forward”)
  - Formally, a set of atomic Boolean **propositions**
    (e.g. {a, b} where a = “velocity of robot > 0.5 m/s” and
    b = “battery is lower than 10% charged”)

Linear Temporal Logic (LTL)

$$\varphi ::= p \mid q \mid \ldots$$ *// atomic propositions*

$$\mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi_1$$

$$\mid \mathbf{G}\varphi_1 \mid \mathbf{F}\varphi_1 \mid \varphi_1 \mathbf{U} \varphi_2 \mid \bigcirc \varphi_1$$

with the following set of **temporal operators**

$\bigcirc$ ("next"), G ("globally"), F ("eventually") and $\mathbf{U}$ ("until").

"The robot will eventually reach its destination"

$$\mathbf{F}(reachDest)$$

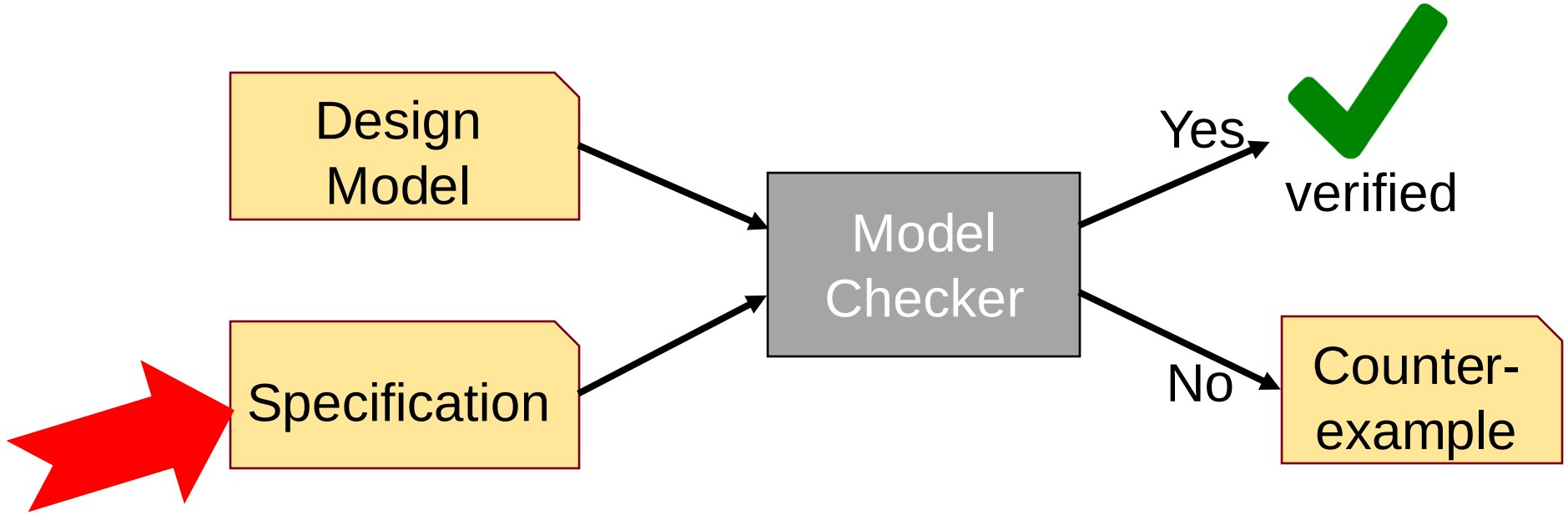
"Every request sent will eventually be served with a response"

$$\mathbf{G}(request \Rightarrow \mathbf{F}(response))$$

"Once the lecture starts, I will keep talking until it ends"

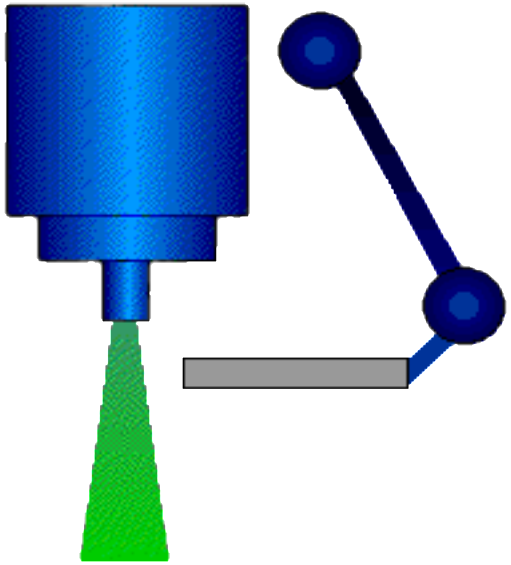
$$startLecture \Rightarrow talk\ \mathbf{U}\ endLecture$$

Specification for Therac-25



- **Specification**: A formal statement of what it means for the design to be "correct"
- **Q. In Therac-25, what is the specification that we want for safety?**

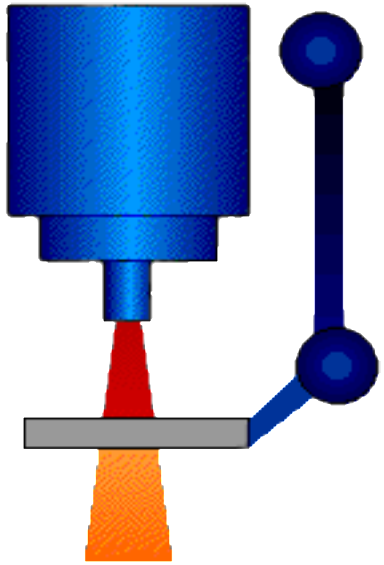
Safety Specification for Therac-25



Electron Mode

Low power

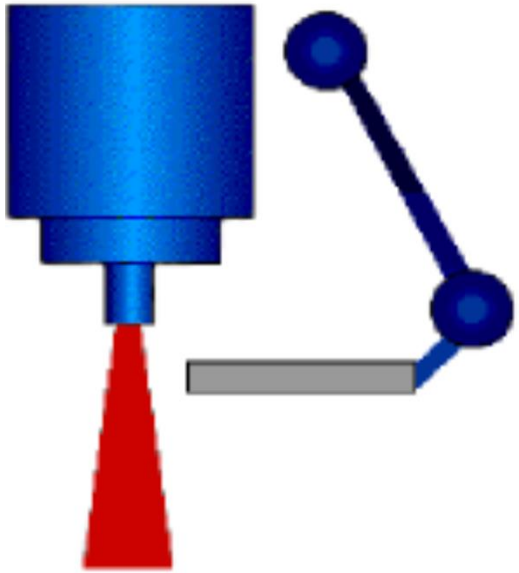
Wide area



X-Ray Mode

High power

Narrow area



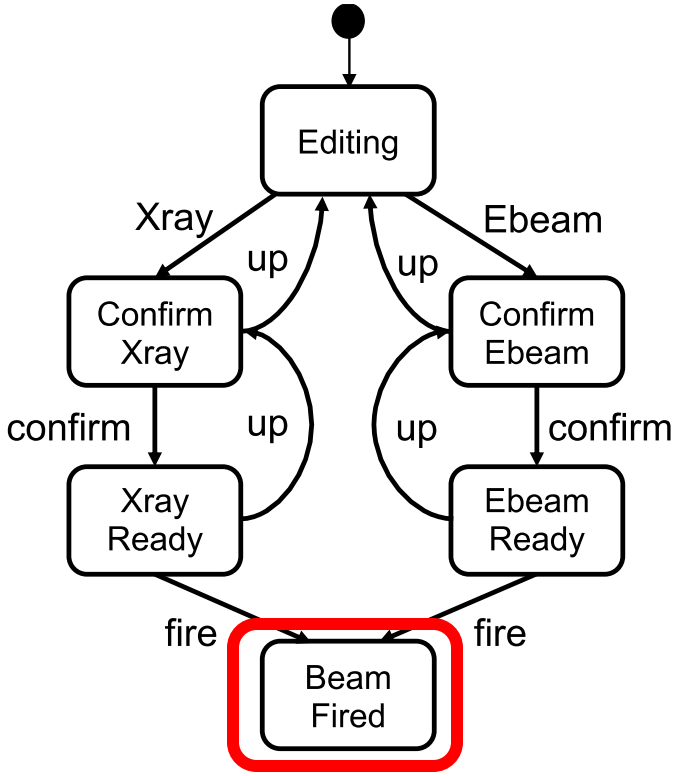
THE PROBLEM

High power

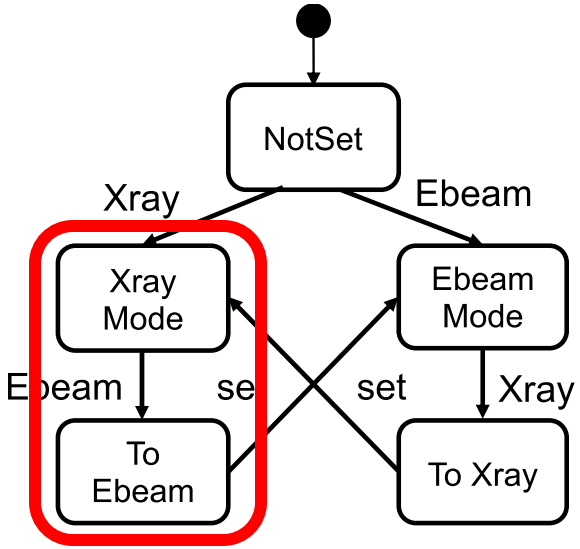
Wide area

- “If the beam is in X-ray mode, it should never be fired with the shield out of place”

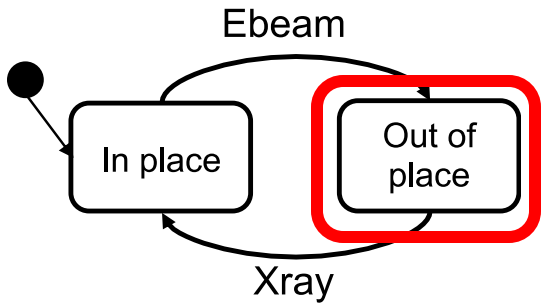
Safety Specification for Therac-25



Interface M_I



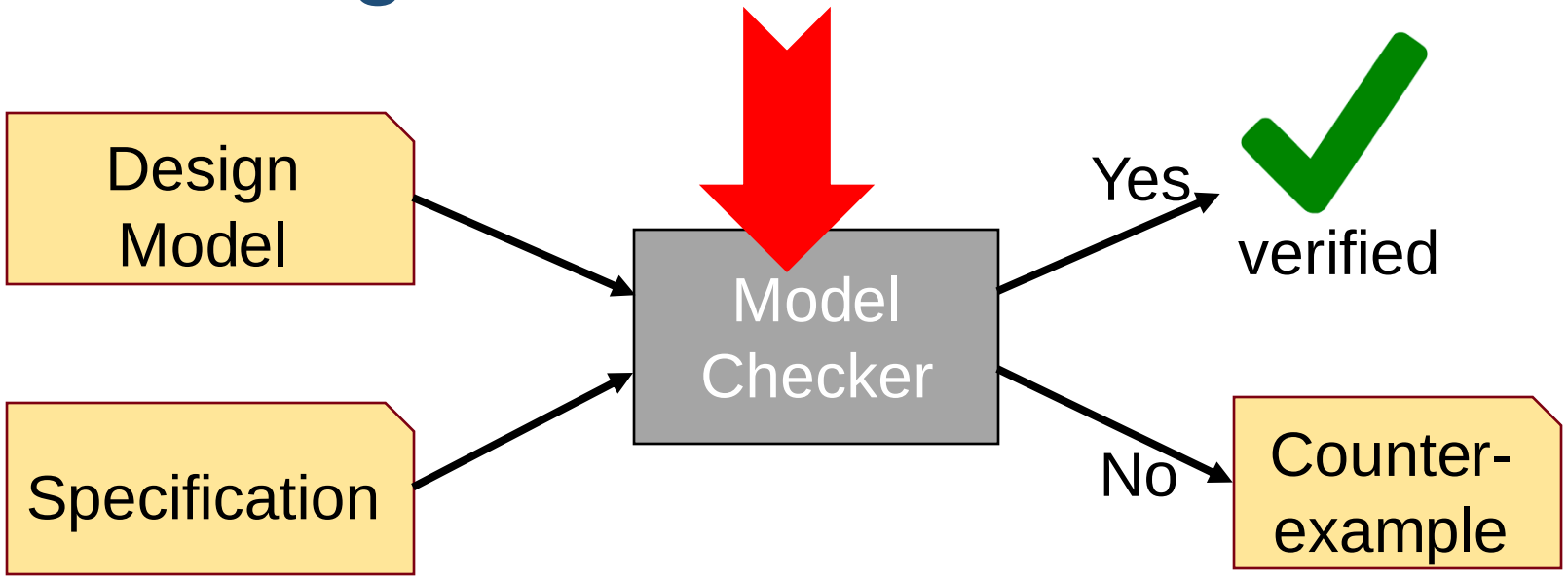
Mode Setter M_B



Spreader M_S

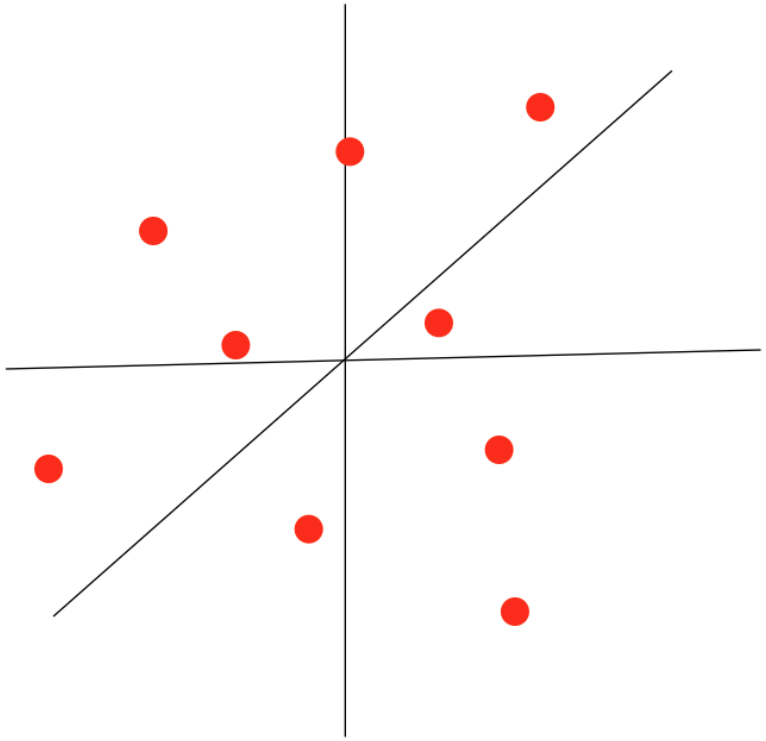
- "If the beam is in X-ray mode, it should never be fired with the shield out of place"
- In temporal logic:
  Globally (M_I = BeamFired /\ $M_B \in$ {XrayMode, ToEbeam} $\Longrightarrow$ M_S != OutOfPlace)

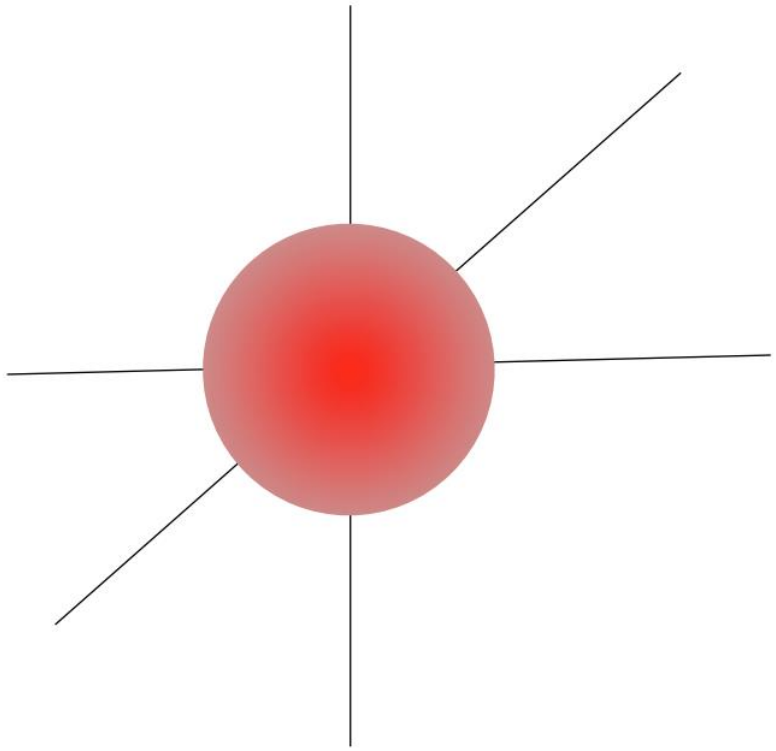
Model Checking



- Given a design model and a specification, the model checker will:
  - Exhaustively explore **all possible traces** in the state machine
  - Look for a trace that leads to a violation of the specification (**counterexample**)
  - If no such trace exists, conclude that the **model satisfies the specification**
- There are many different algorithms and tools for model checking

Testing vs. Model Checking





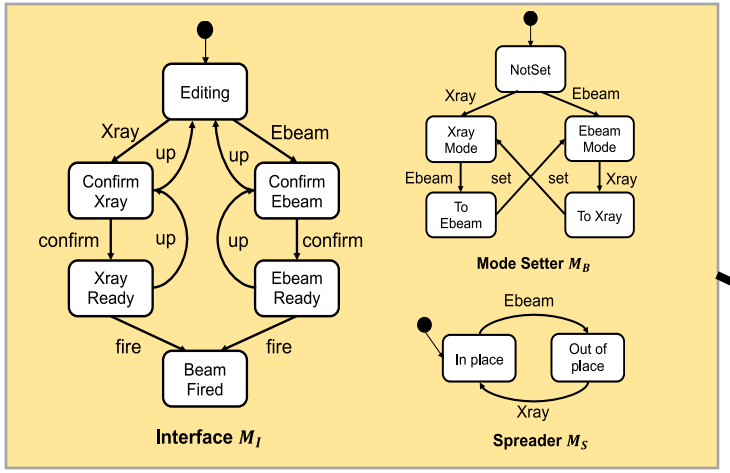
Testing

- Manually create test cases
- Difficult to cover all corner cases
- Cannot prove absence of bugs

Model Checking

- No need to create tests
- Covers all possible executions
- If bug exists, it’s guaranteed to find it

Therac-25: Putting it all together



Design model

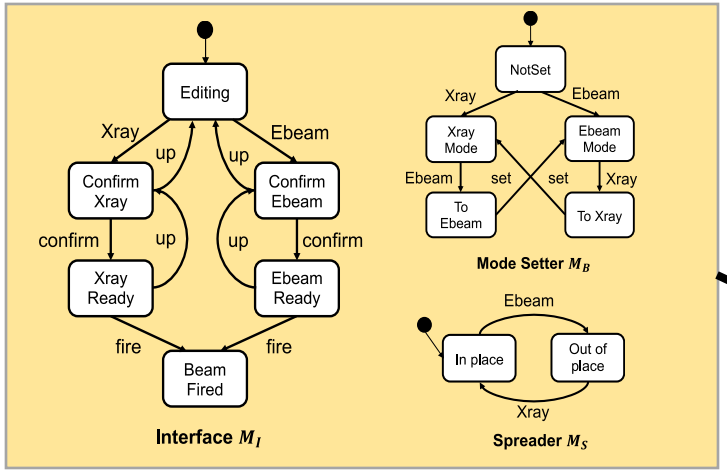
$!(M_I = BeamFired \wedge M_B \in \{XrayMode, ToEbeam\} \wedge M_S = OutOfPlace)$

Specification

Model Checker

Does the design satisfy the specification? Yes/No?

Therac-25: Putting it all together



Design model

!(M_I = BeamFired /\
M_B ∈ {XrayMode, ToEbeam} /\
M_S = OutOfPlace)

Specification

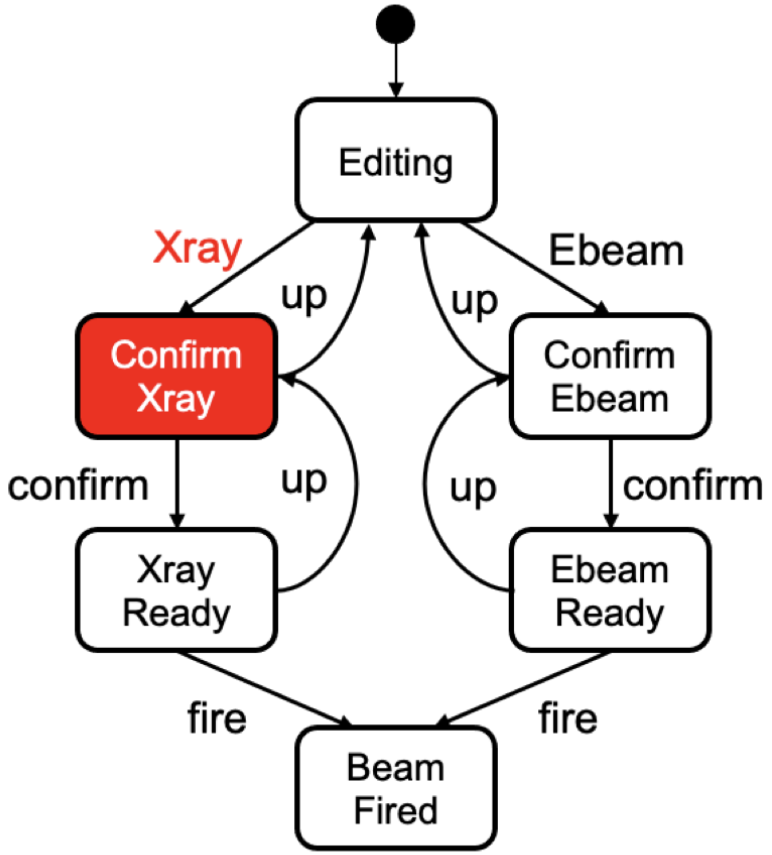
Model Checker

No

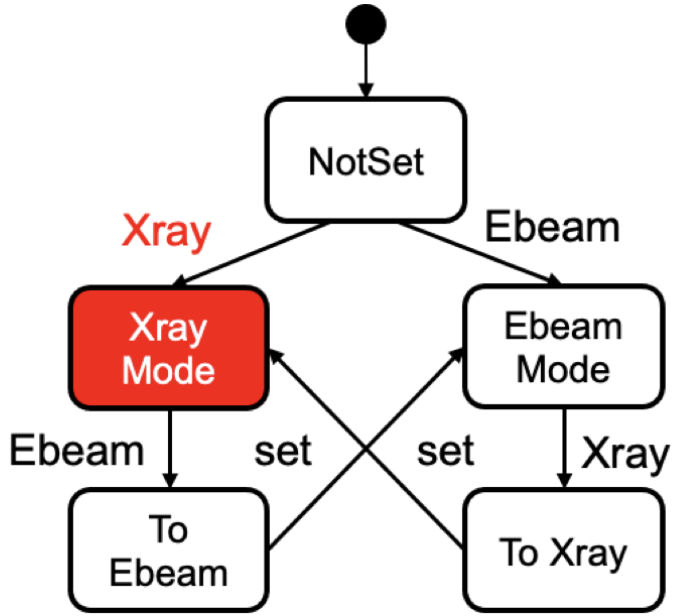
Counterexample found!

Counterexample

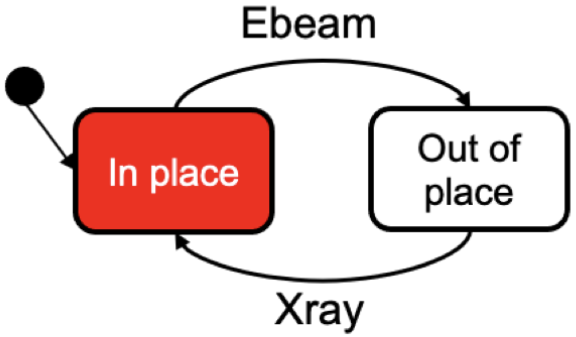
- Operator wants E-beam, but selects X-ray by mistake
- System is in X-ray mode



Interface M_I



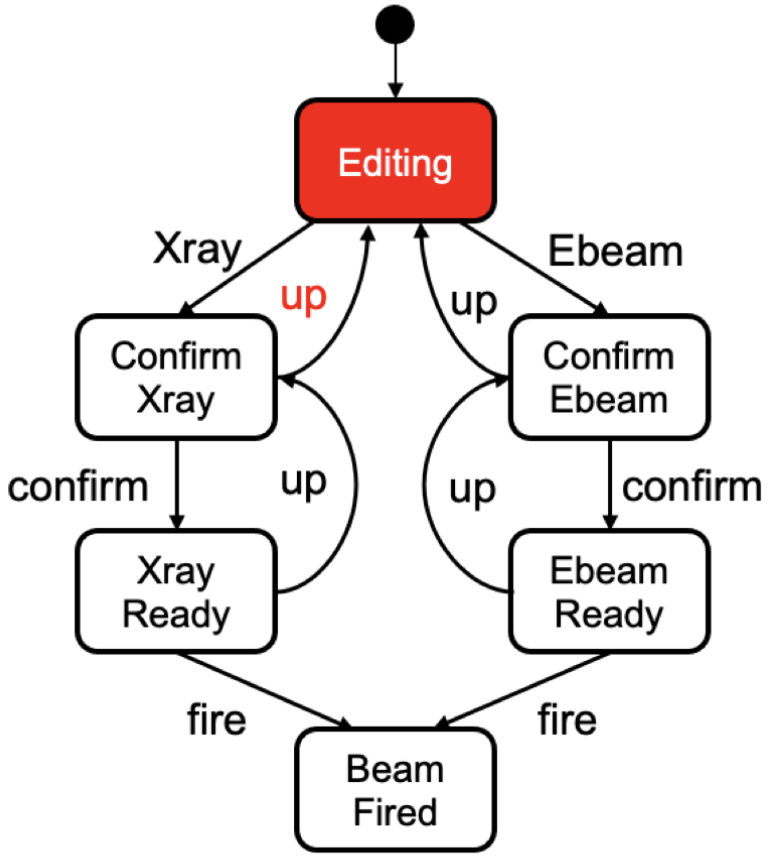
Mode Setter M_B



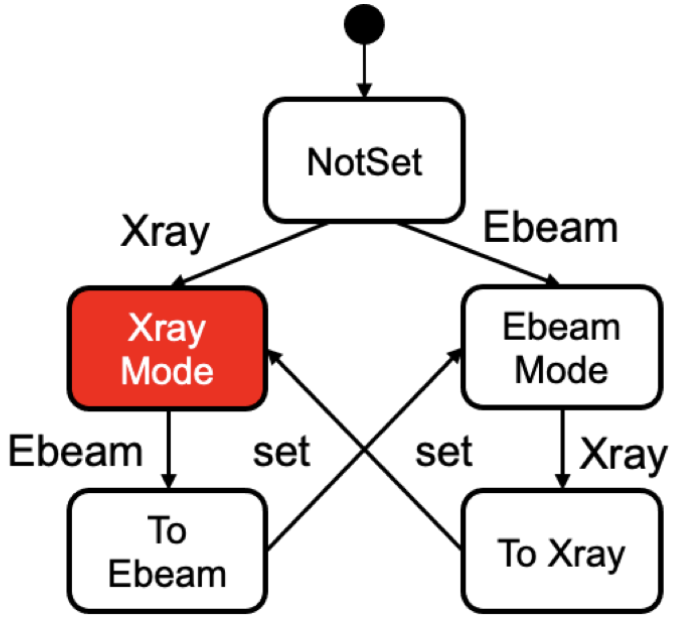
Spreader M_S

Counterexample

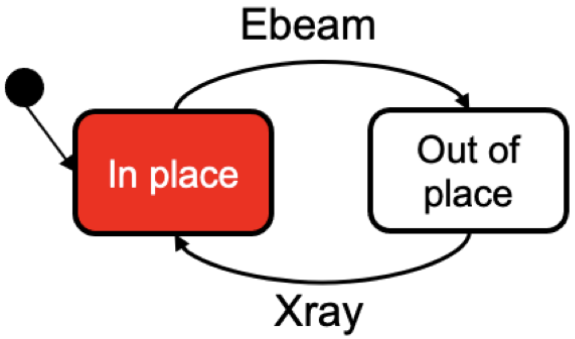
Operator realizes mistake and goes back to the setting



Interface M_I



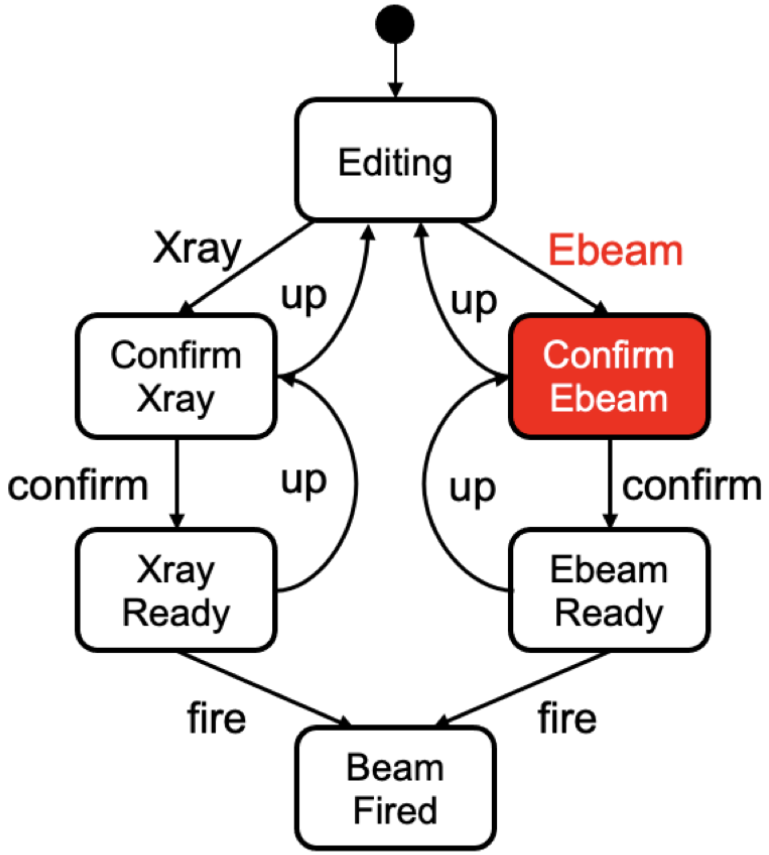
Mode Setter M_B



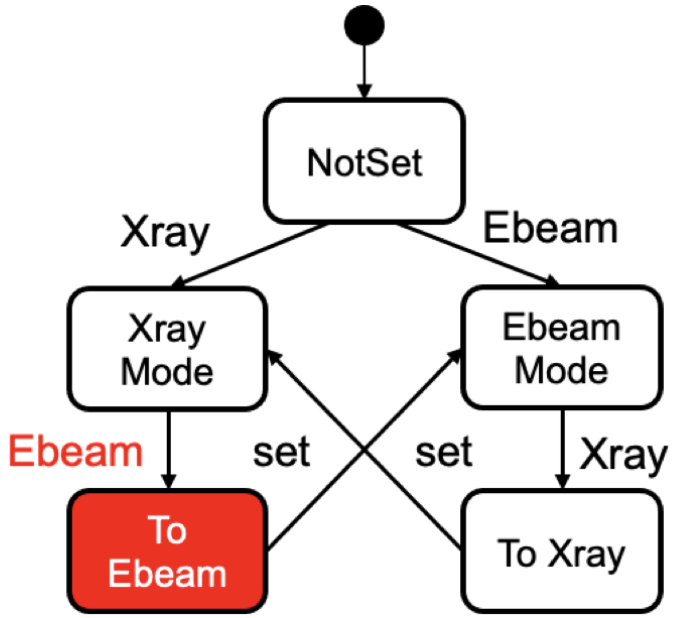
Spreader M_S

Counterexample

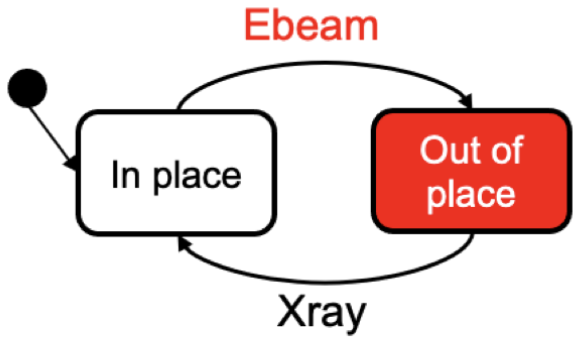
- Operator selects E-beam
- Shield is removed



Interface M_I



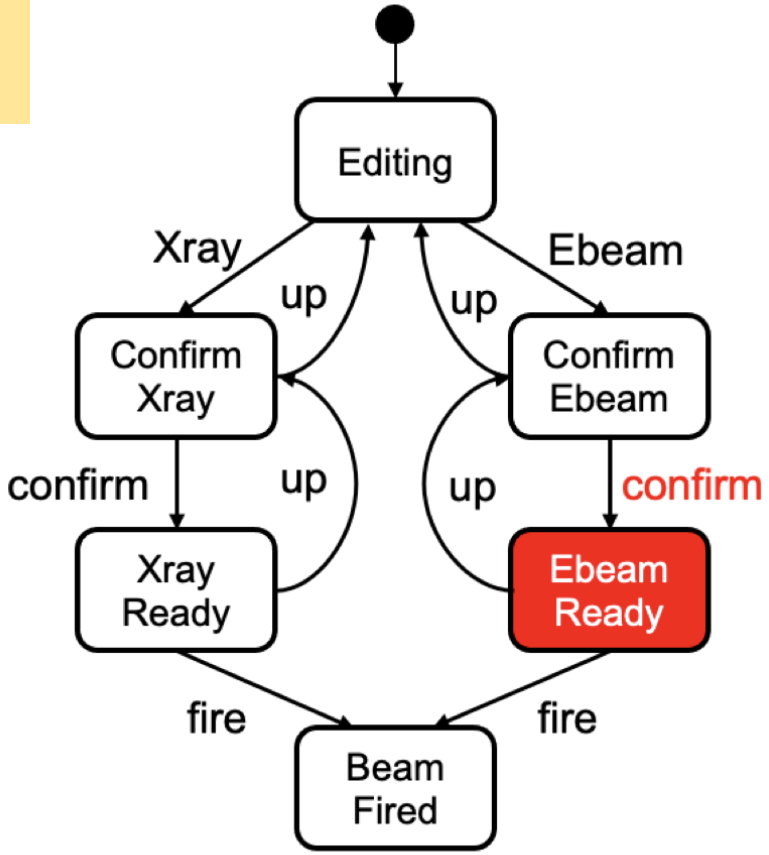
Mode Setter M_B



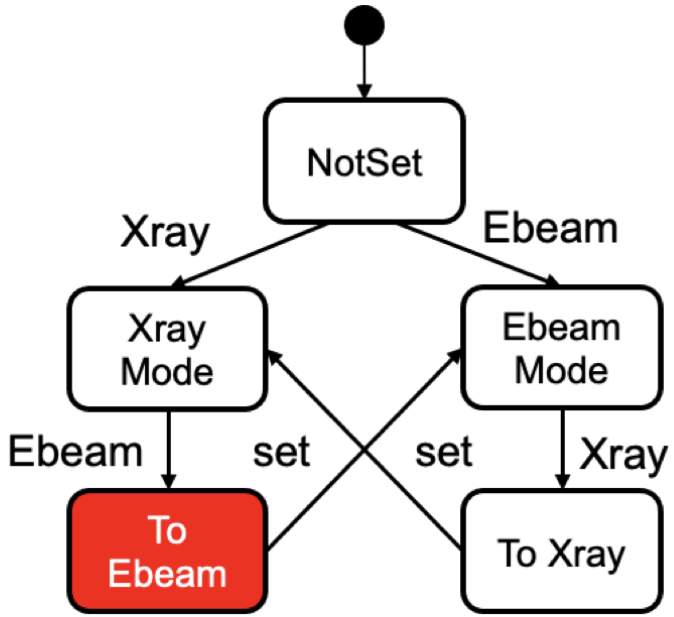
Spreader M_S

Counterexample

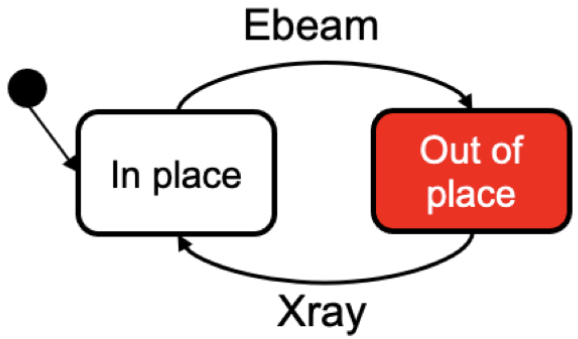
System is still switching from X-ray to Ebeam



Interface M_I



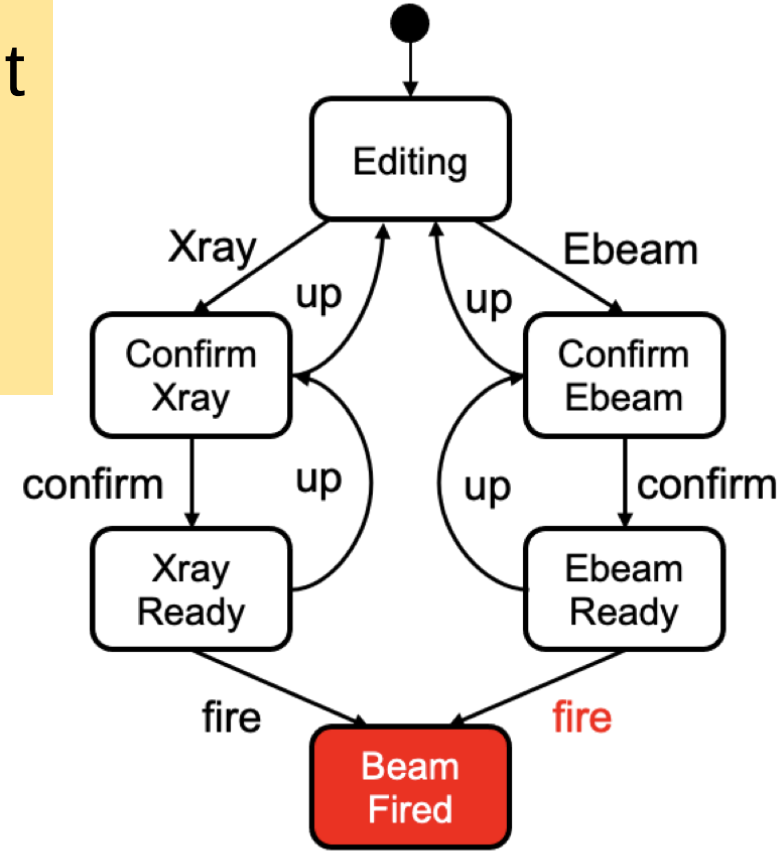
Mode Setter M_B



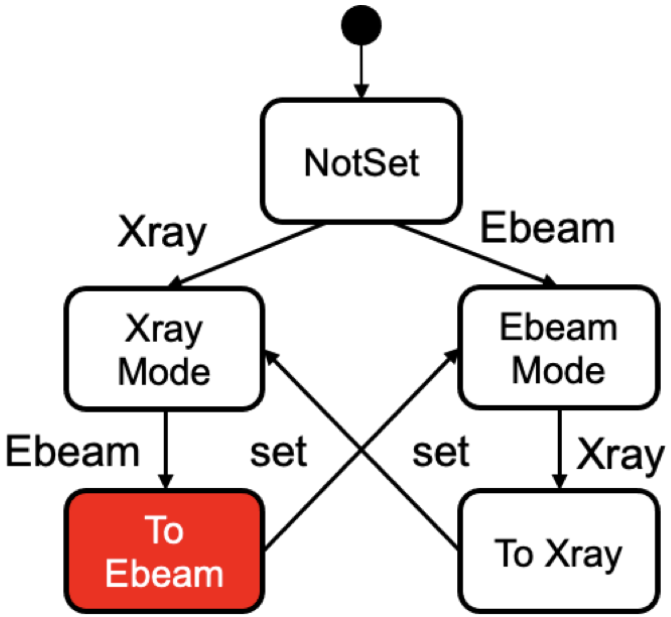
Spreader M_S

Counterexample

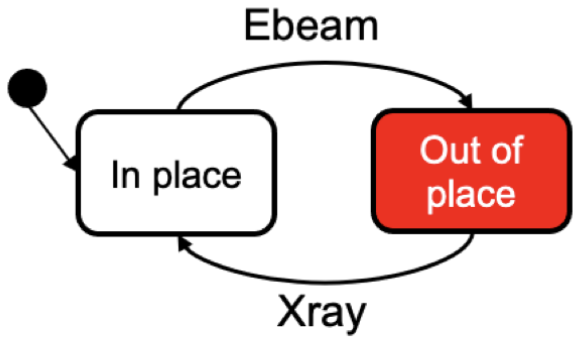
- The beam is fired in X-ray mode with shield out
- Safety violation! This causes radiation overdose



Interface M_I

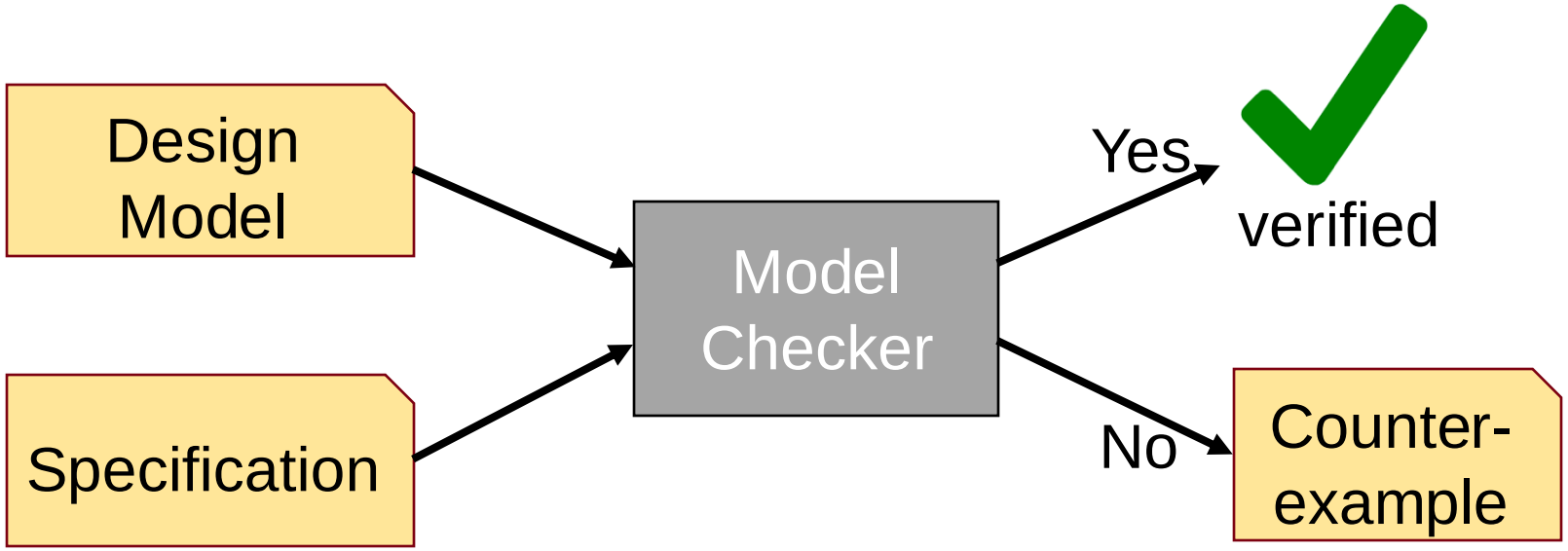


Mode Setter M_B



Spreader M_S

Summary: Model Checking



- An approach for automatically checking software design to find errors
- Input
  - **Design model**: A formal, mathematical model of a system design
  - **Specification**: A formal statement of what it means for the design to be “correct”
- Output
  - A **counterexample** that demonstrates how the system fails its specification

Industrial Application: Microsoft SLAM Project

- Device driver bugs were one of the leading causes of Windows crashes (85% in Windows XP)
- Those bugs involve incorrect usage of the Windows API for accessing critical OS resources

Microsoft SLAM Project

- **Goal**: Automatically analyze the device drivers and use **model checking** to find potential bugs
- Automatically analyze the source code (in C) to extract **state machines** that describe its high-level design
- Formalize safe API use rules as the correctness **specification**
- Highly successful; found hundreds of bugs across many device drivers
- SLAM is now distributed to driver developers as part of the Windows Driver Foundation
- SLAM is known to be one of the most successful applications of formal methods in industry

Model Checking: Discussion

- Model checking is a complementary approach to testing
  - Unlike testing, it automatically searches all possible executions for bugs
  - It there's a bug, it is guaranteed to find it!
- However, model checking is **NOT** the perfect solution to ensuring software quality
  - **Q. What are some limitations/challenges with model checking?**

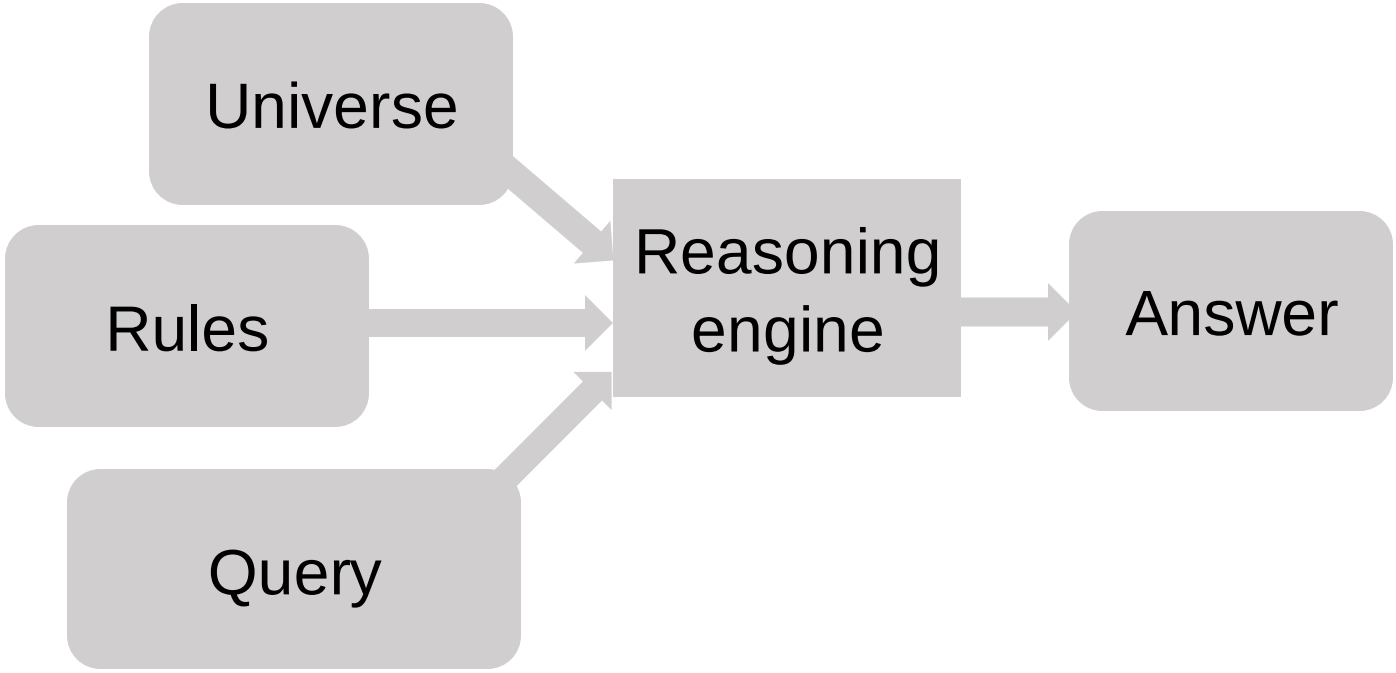
Model Checking: Discussion

- Model checking is a complementary approach to testing
  - Unlike testing, it automatically searches all possible executions for bugs
  - It there's a bug, it is guaranteed to find it!
- However, model checking is **NOT** the perfect solution to ensuring software quality
  - Analysis is done over a **model** of the system design
    - If the model is inaccurate, you may miss bugs
    - The model can be very large (e.g., billions of states) and take a long time
  - You need to provide a **formal specification of correctness**
    - If the specification is incorrect, you may also miss bugs
    - For some systems, it's really hard to say what "correctness" means **(Q. any examples?)**

Formal Methods

- A class of techniques for ensuring software quality
- **Goal**: Provide strong, **mathematical** guarantees about the properties or behavior of software
- Types of formal methods:
  - Model checking
  - **Automated reasoning**
  - Interactive theorem proving
- Different methods, different levels of automation and guarantees provided
- A wide range of applications: Security analysis, bug finding, configuration analysis, program synthesis, etc.,

Automated Reasoning



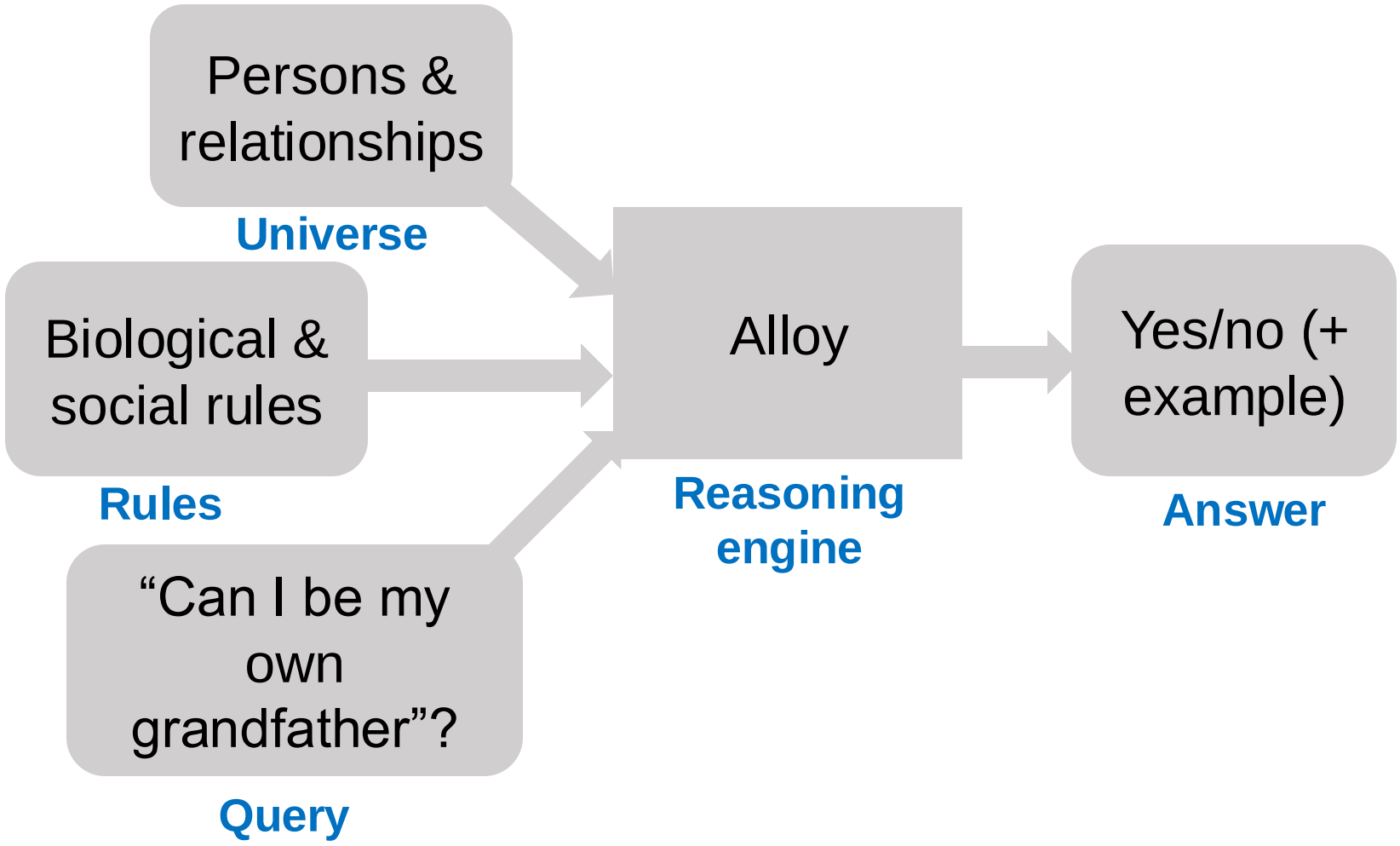
- **Universe**: A set of concepts (objects & relationships) that you are reasoning about
- **Rules** that must be followed by the concepts in the universe
- **Query**: A question that you'd like to ask about the universe
- **Answer** to the query, computed by the **reasoning engine**

Riddle

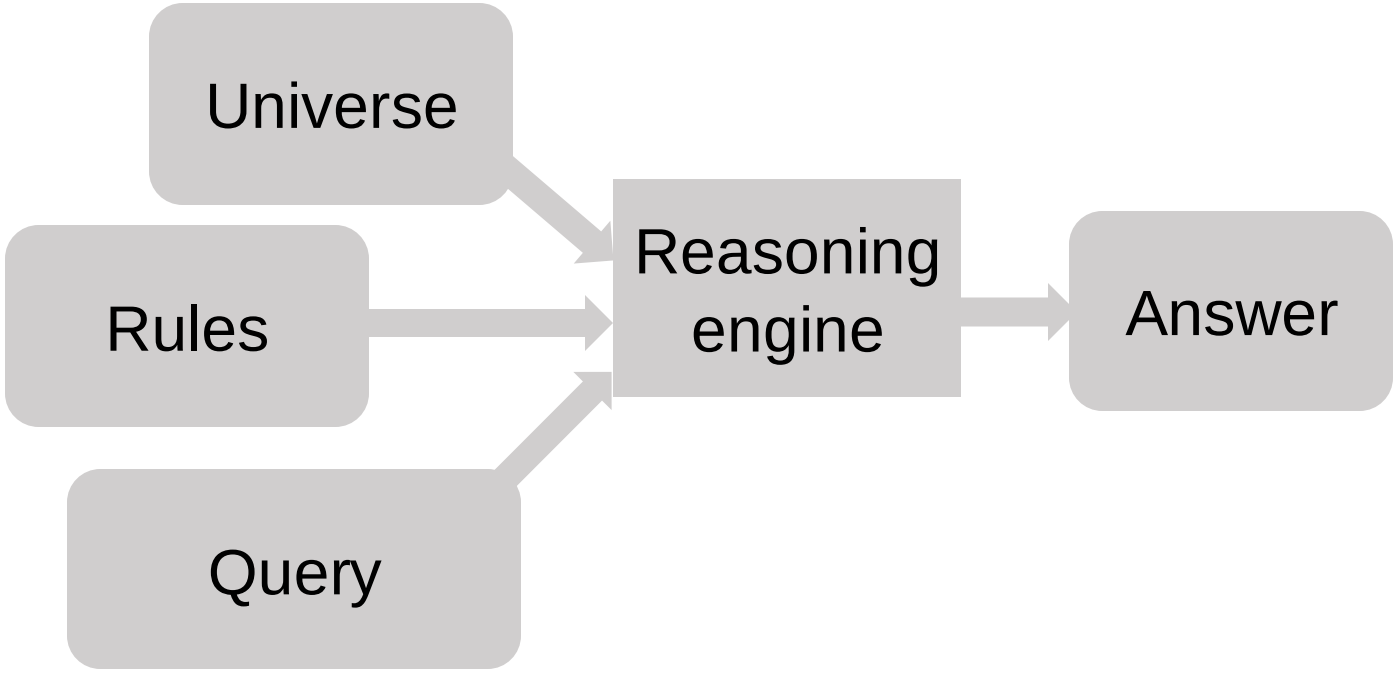
- Is it possible for someone to be their own grandfather?

Demo: My Own Grandpa

My Own Grandpa

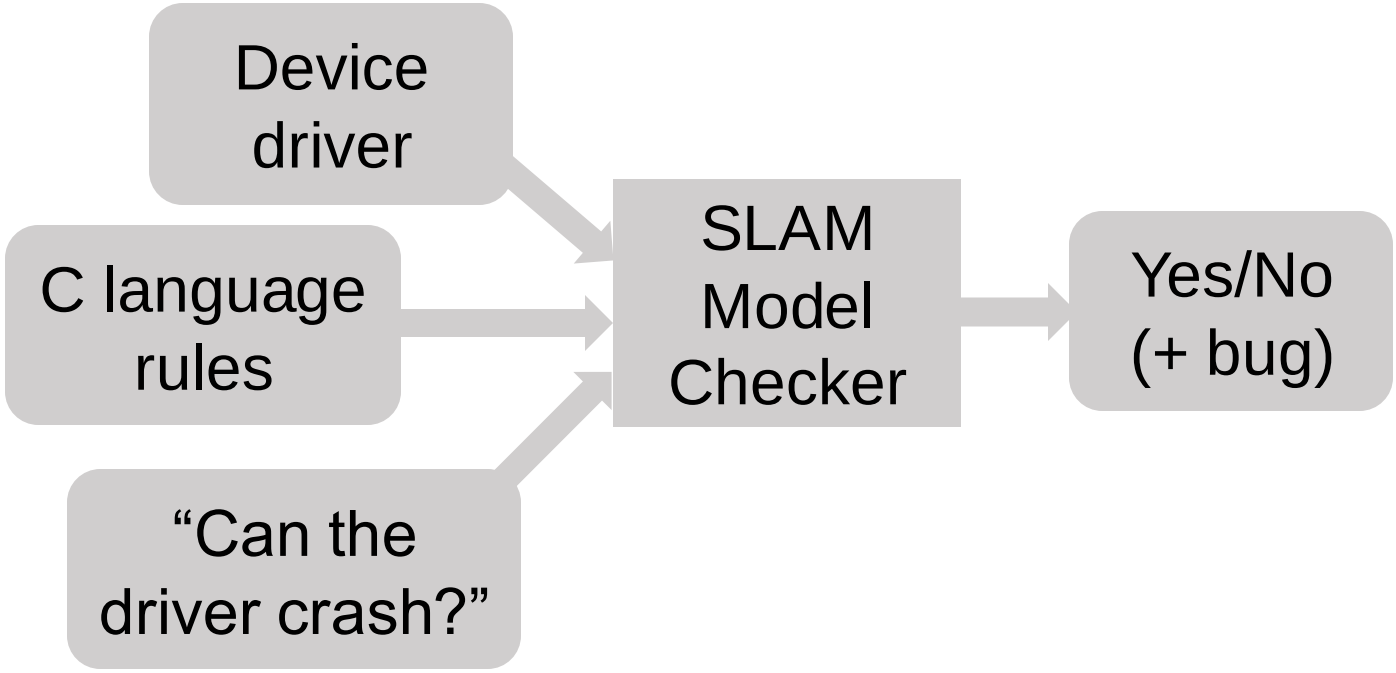


Automated Reasoning



- Many quality assurance tasks in software engineering can be formulated as a type of automated reasoning problem!

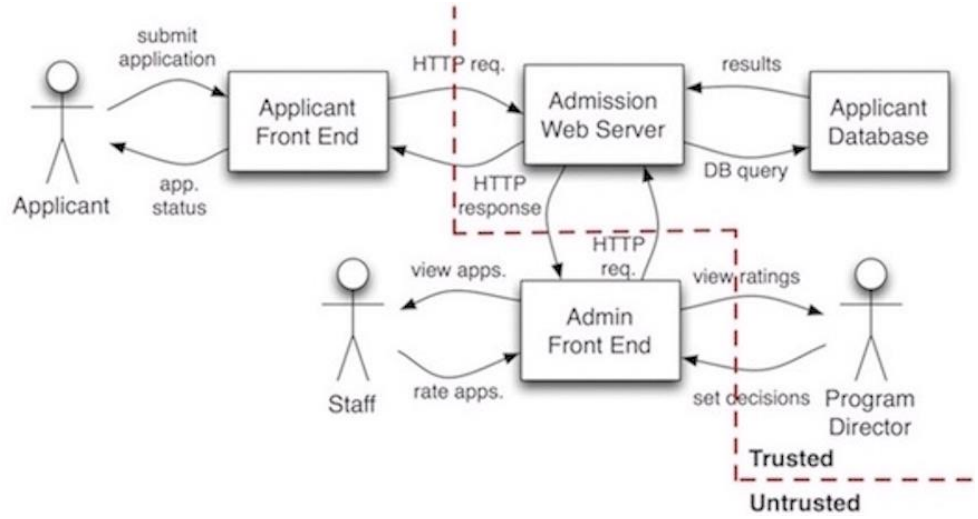
SLAM Device Driver Analyzer@Microsoft



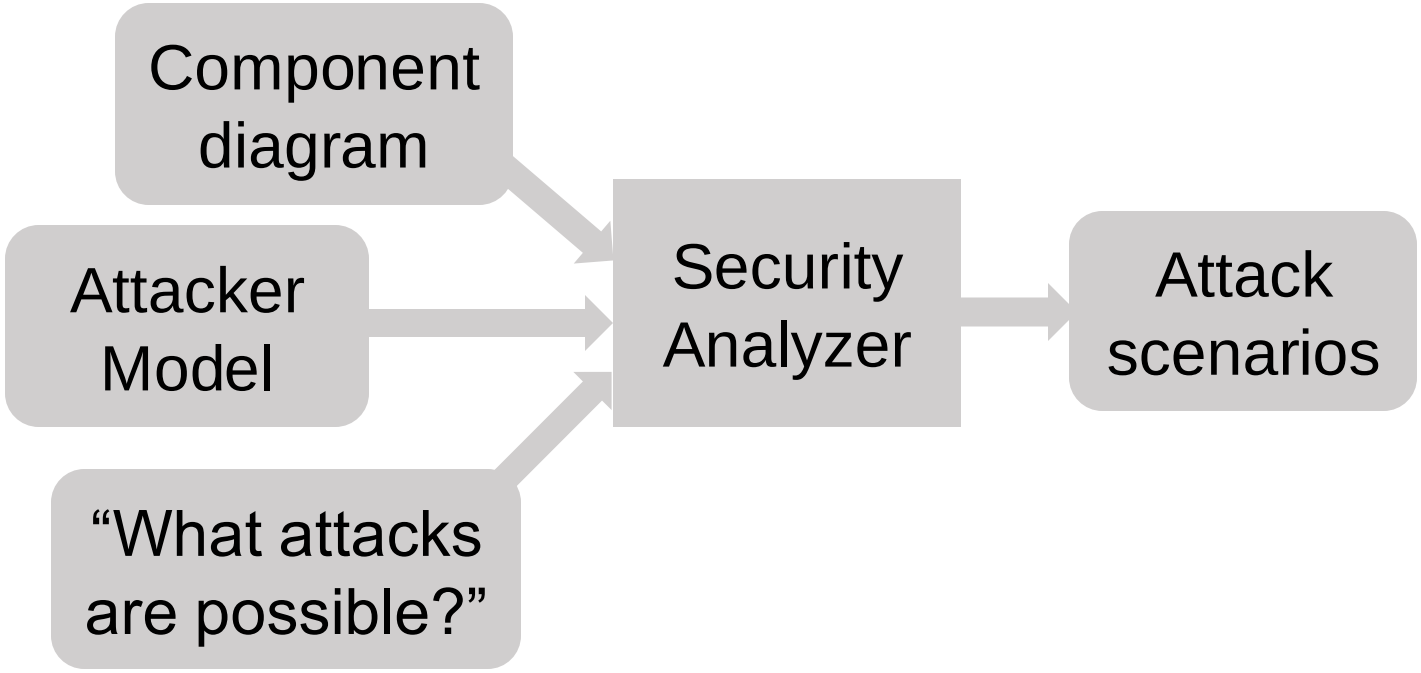
- Model checking itself can be formulated a type of automated reasoning
- **Query**: “Does there exist a system execution that violates a specification?”

Recall: Threat modeling (from security lectures)

Threat	Desired property	Threat Definition
Spoofing	Authenticity	Pretending to be something or someone other than yourself
Tampering	Integrity	Modifying something on disk, network, memory, or elsewhere
Repudiation	Non-repudiability	Claiming that you didn't do something or were not responsible; can be honest or false
Information disclosure	Confidentiality	Someone obtaining information they are not authorized to access
Denial of service	Availability	Exhausting resources needed to provide service
Elevation of privilege	Authorization	Allowing someone to do something they are not authorized to do



Security Analysis



- Given a component diagram specified in a machine-readable language, the process threat modeling can be automated
- **Security analyzer**: Simulates every possible behavior of the attacker to find possible attacks (if any exists)

Example: Spectre & Meltdown

ANDY GREENBERG SECURITY 05.14.2019 01:00 PM

Meltdown Redux: Intel Flaw Lets Hackers Siphon Secrets from Millions of PCs

Two different groups of researchers found another speculative execution attack that can steal all the data a CPU touches.

- Found by Google researchers in 2017
- Allows a malicious program to read all memory
- Affected Intel x86, AMD, ARM, IBM processors
- Caused Intel to redesign its processors

Automatically Finding Spectre-like Attacks

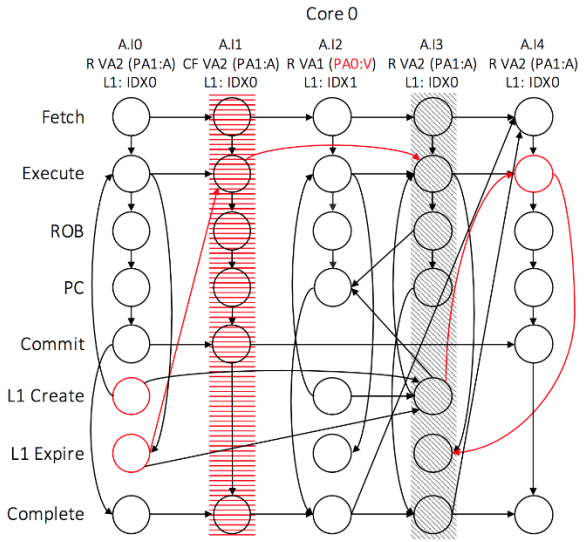
CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests

Caroline Trippel
Princeton University
ctrippel@princeton.edu

Daniel Lustig
NVIDIA
dlustig@nvidia.com

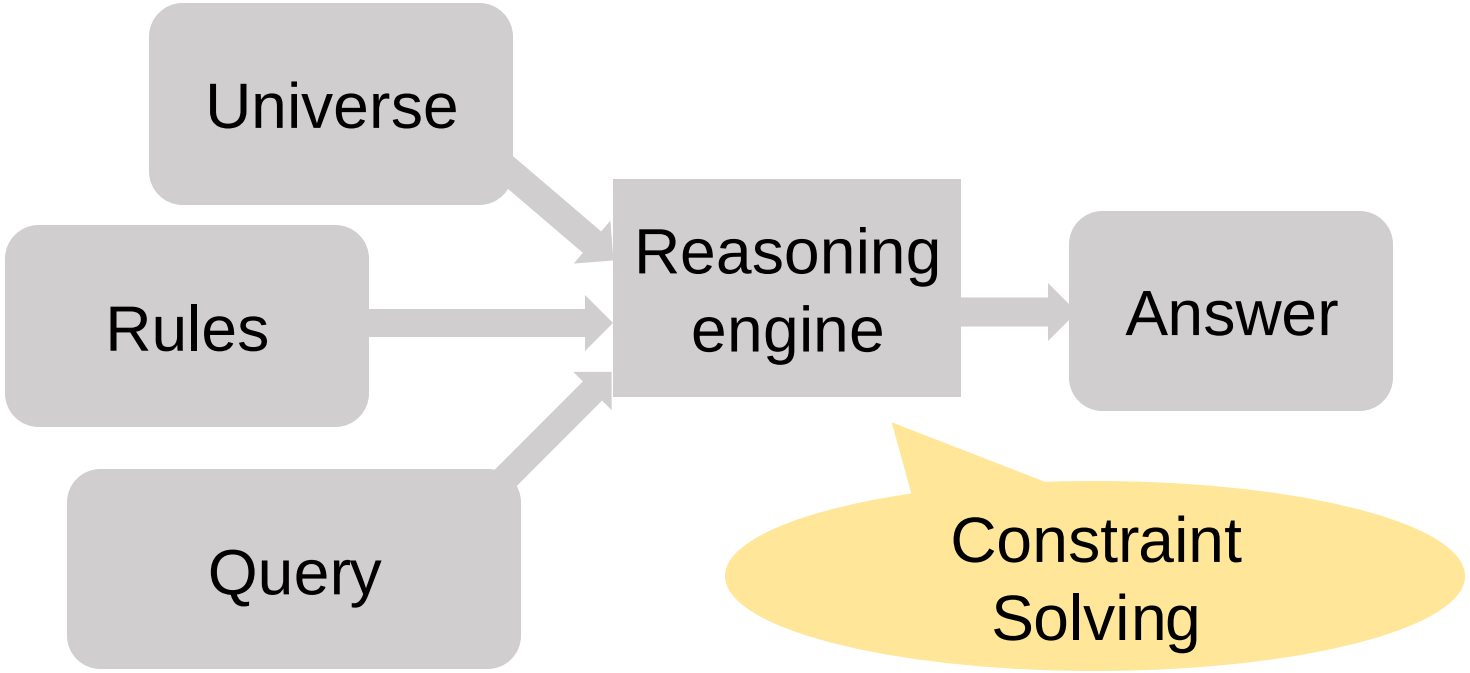
Margaret Martonosi
Princeton University
mrm@princeton.edu


```
sig Address { }
abstract sig Event { po: lone Event }
abstract sig MemoryEvent extends Event { address: one Address }
sig Write extends MemoryEvent { rf : set Read, co : set Write }
sig Read extends MemoryEvent { fr : set Write }
fun com : MemoryEvent->MemoryEvent { rf + fr + co }
abstract sig Location { }
sig Node {
  event: one Event,
  loc: one Location,
  uhb: set Node
}
```



(a) Meltdown

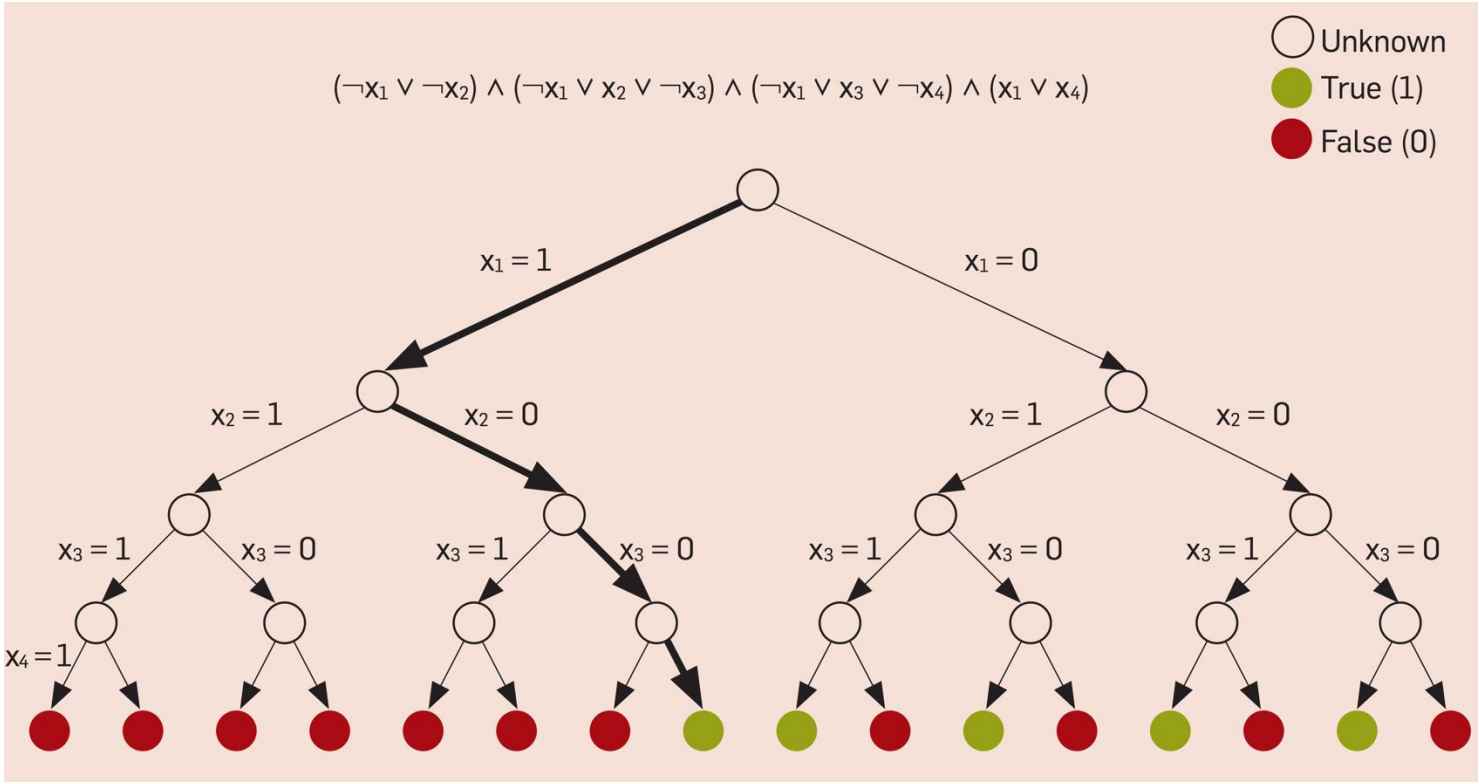
Automated Reasoning as Constraint Solving



How is the automatic reasoning typically done?

- By solving a **constraint satisfaction problem**

Boolean Satisfiability Problem (SAT)



- A common type of constraint satisfaction problem
- All variables are propositions (0 or 1), with basic Boolean operators
- Given **N** variables, there are $\mathbf{2^N}$ possible assignments

Industrial Uses of Automated Reasoning

AWS Verified Access | Overview | Features | Pricing | Partners | Customers

« Networking

AWS Verified Access

Provide secure access to corporate applications without a VPN

Get started with Verified Access

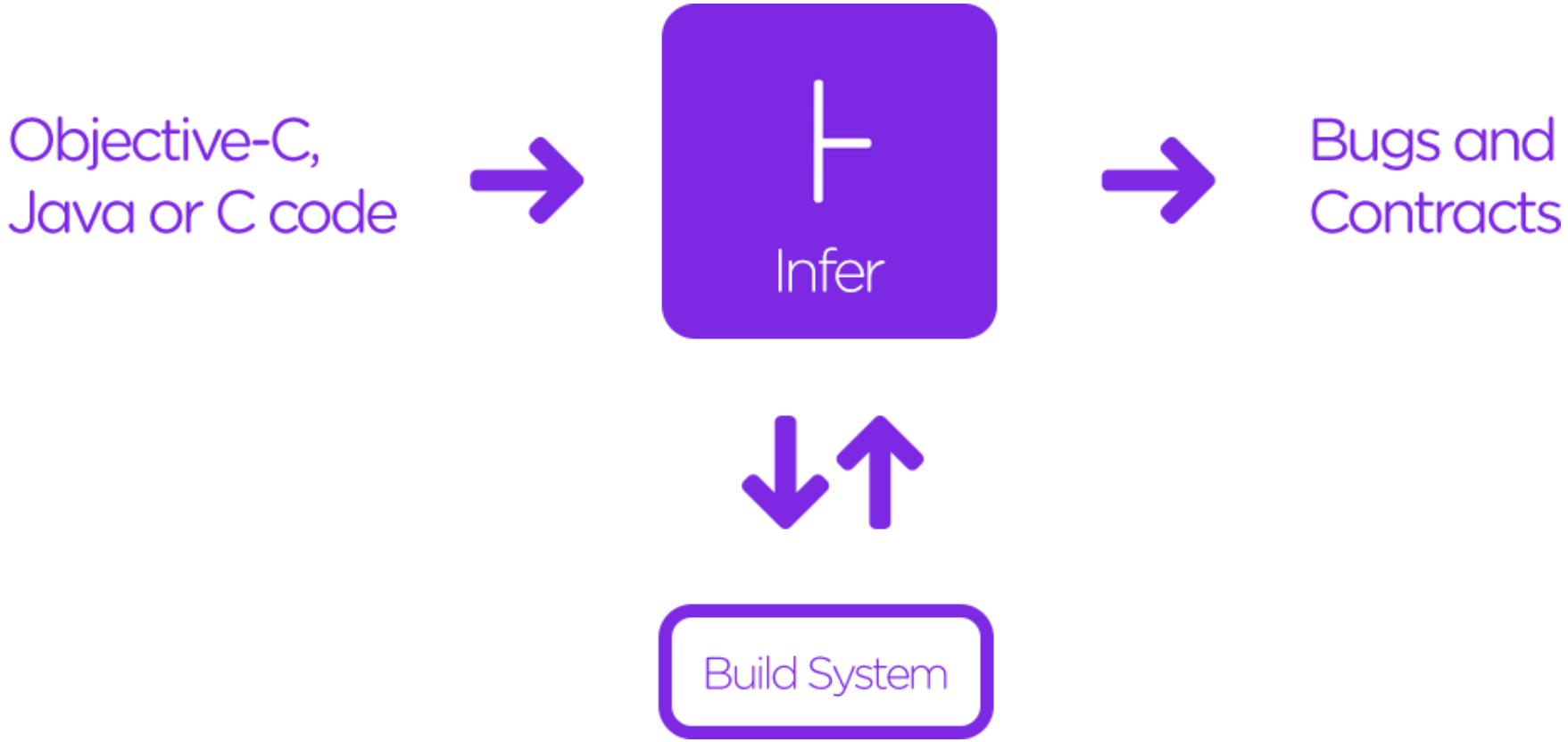
Improve security posture by evaluating each access request in real time against predefined requirements.

Deliver a seamless user experience through virtual access to corporate applications without a VPN.

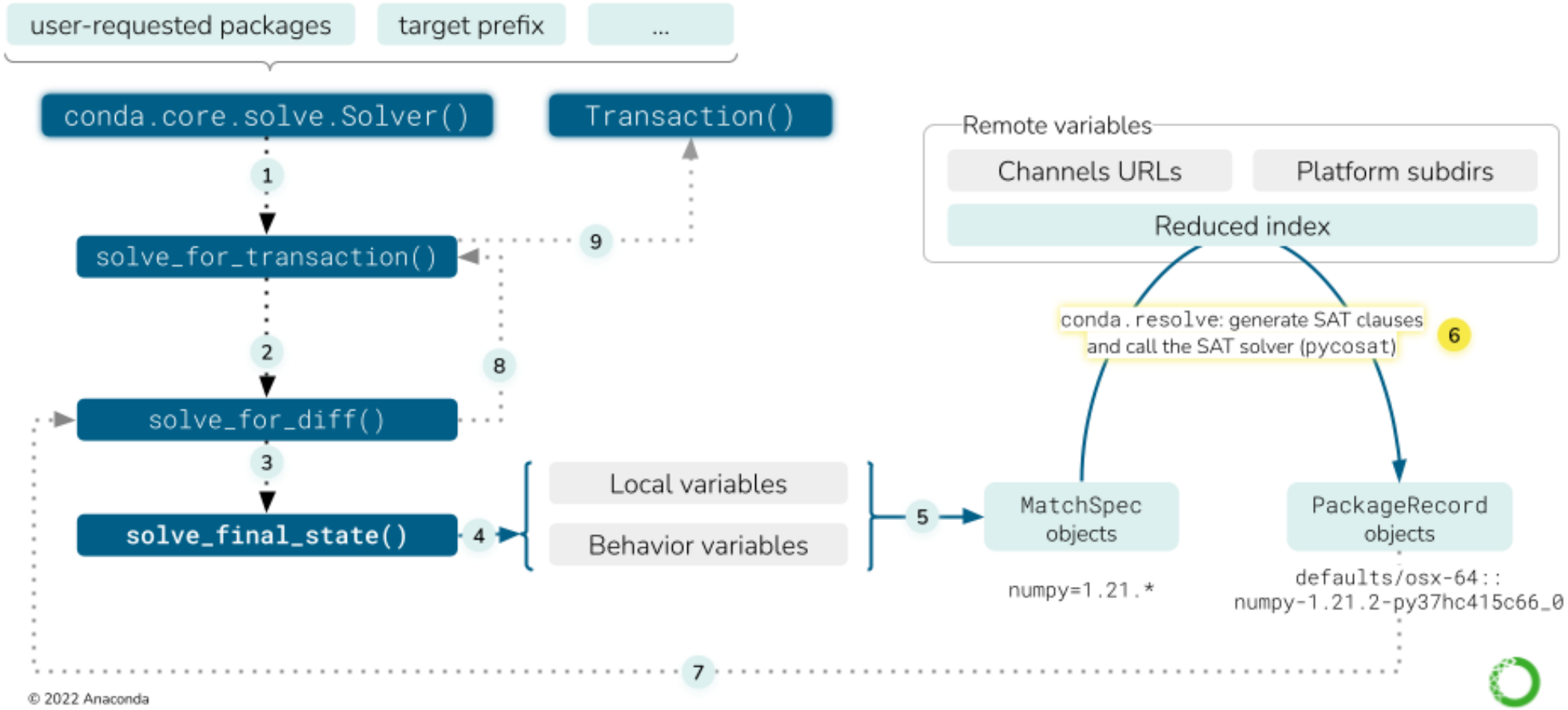
Define a unique access policy for each application, with conditions based on identity data and device posture.

A tool to detect bugs in Java and C/C++/Objective-C code before it ships

Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs. Anyone can use Infer to intercept critical bugs before they have shipped to users, and help prevent crashes or poor performance.



Inside the Solver: formulating the problem



Dependency Management: Anaconda (Python), apt-get (Debian), Eclipse plugins, pkg (FreeBSD)…

Automated Reasoning: Industry Uses

- **Enterprise software**
  - Microsoft: Finding bugs in Windows OS and device drivers
  - Amazon: AWS security issues
  - Facebook: Continuous integration (CI) analysis
- **Cryptocurrency** (e.g., Ethereum)
  - Verified blockchain transactions
- **Hardware chip design & verification**
  - Apple, Intel, AMD, Samsung
- **Safety-critical systems**
  - NASA, Airbus, Boeing
- …and many others!

Formal Methods: Takeaways

- Formal methods is another approach to evaluating system designs beside testing
- Techniques such as model checking and automated reasoning can provide a much **stronger level of assurance** than testing
- However, these techniques are complementary, not a replacement
  - They typically require (i) a formal model of the system and (ii) a formal specification that describes the “correctness” condition
  - If either one of these is wrong, then guarantees provided by these techniques are not meaningful
- Automated reasoning is increasingly used in industry to improve QA
- Formal methods is an active area of research; there will be new, powerful tools and techniques available

Summary

- Exit ticket!