Input-based Three-Valued Abstraction Refinement

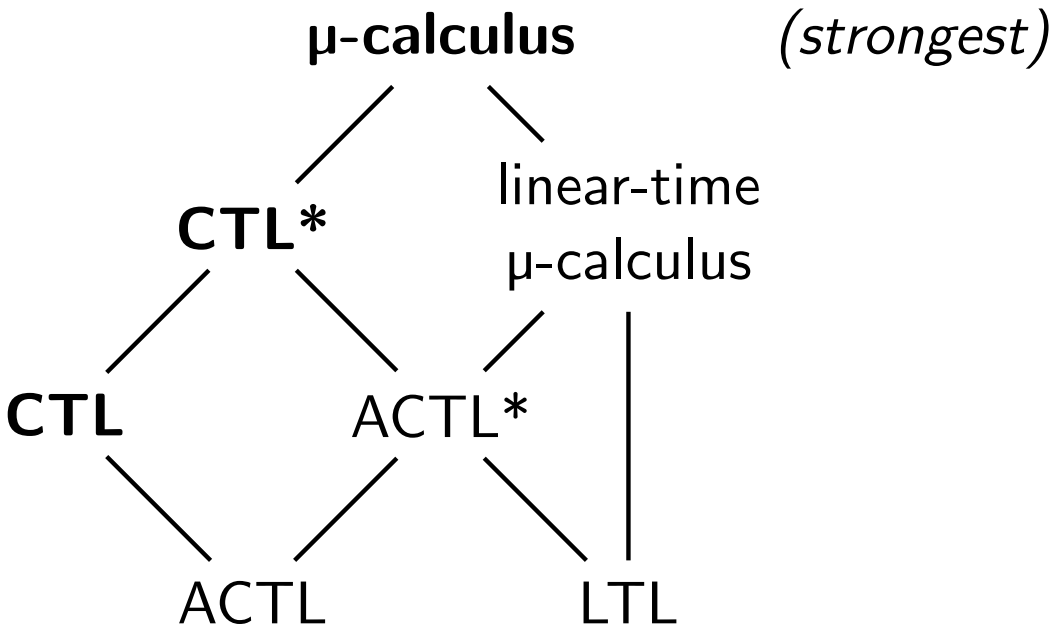
Jan Onderka[1,2,3] & Stefan Ratschan[2]

[1] Faculty of Engineering, University of Freiburg
[2] Institute of Computer Science, The Czech Academy of Sciences
[3] Faculty of Information Technology, Czech Technical University in Prague

[1]onderka@cs.uni-freiburg.de

2026-01-13

Temporal logics for model checking



- Classic model checking: can verify all properties of μ-calculus
- Scales poorly unless *abstraction refinement* is used
- Usually: Counterexample-guided Abstraction Refinement (CEGAR)
  - Huge success for path-universal logics
  - Cannot verify logics in **bold**
  - Should we care about this?

Motivation: recovery properties 1/2

- Long-running (embedded/CLI/GUI...) program with main loop:

```
fn main() {
    ...
    loop {
        // wait until the user presses a button
        while (button_not_pressed()) {}
        // perform some action
        label: do_something();
        ...
    }
}
```

- Possible bug: infinite loop somewhere in the program

```
if complicated_prerequisites() {
    loop {}
}
```

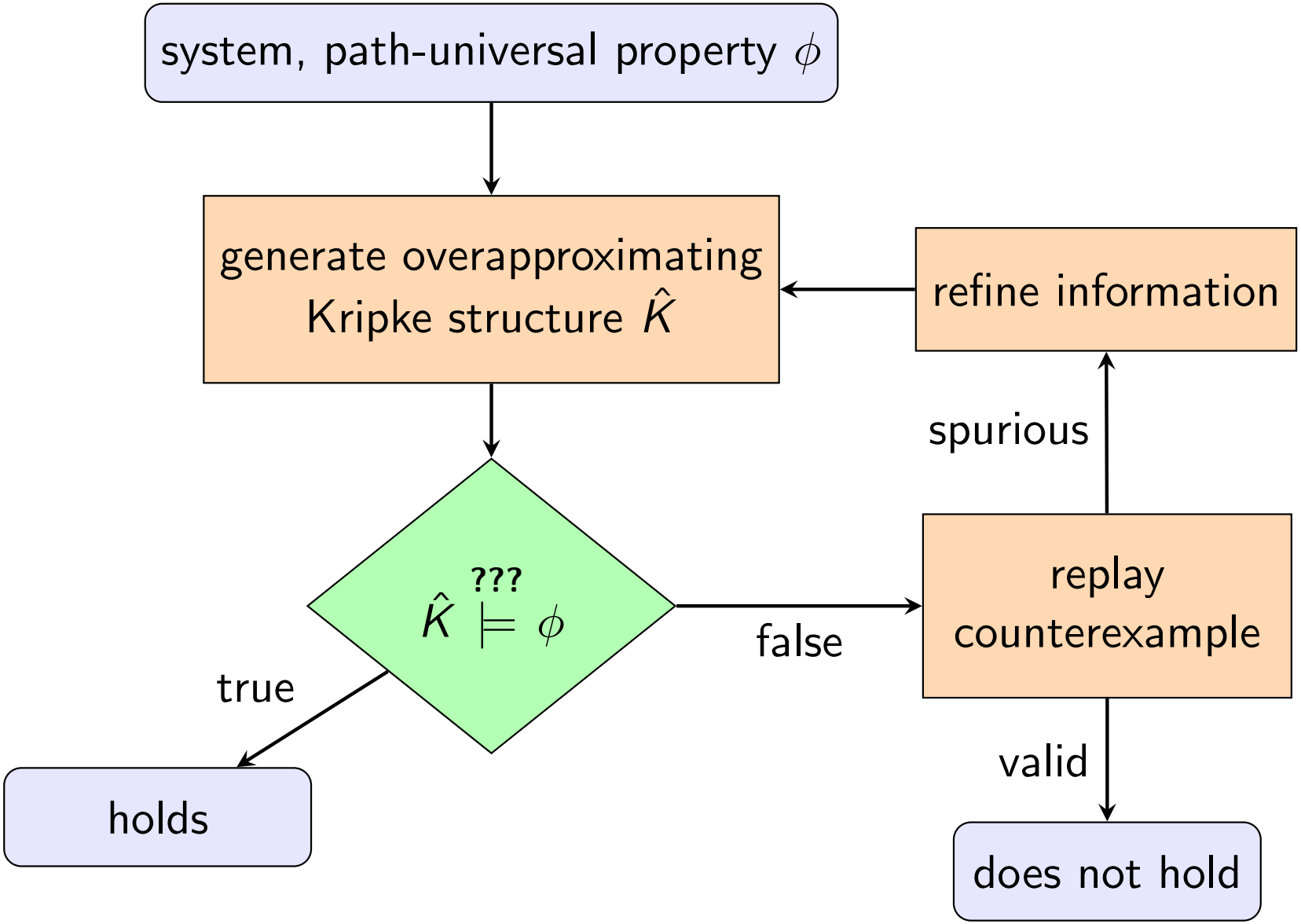
- We want this (intuitively): "no matter where we are in the program, we can reach `label` somehow"

Motivation: recovery properties 2/2

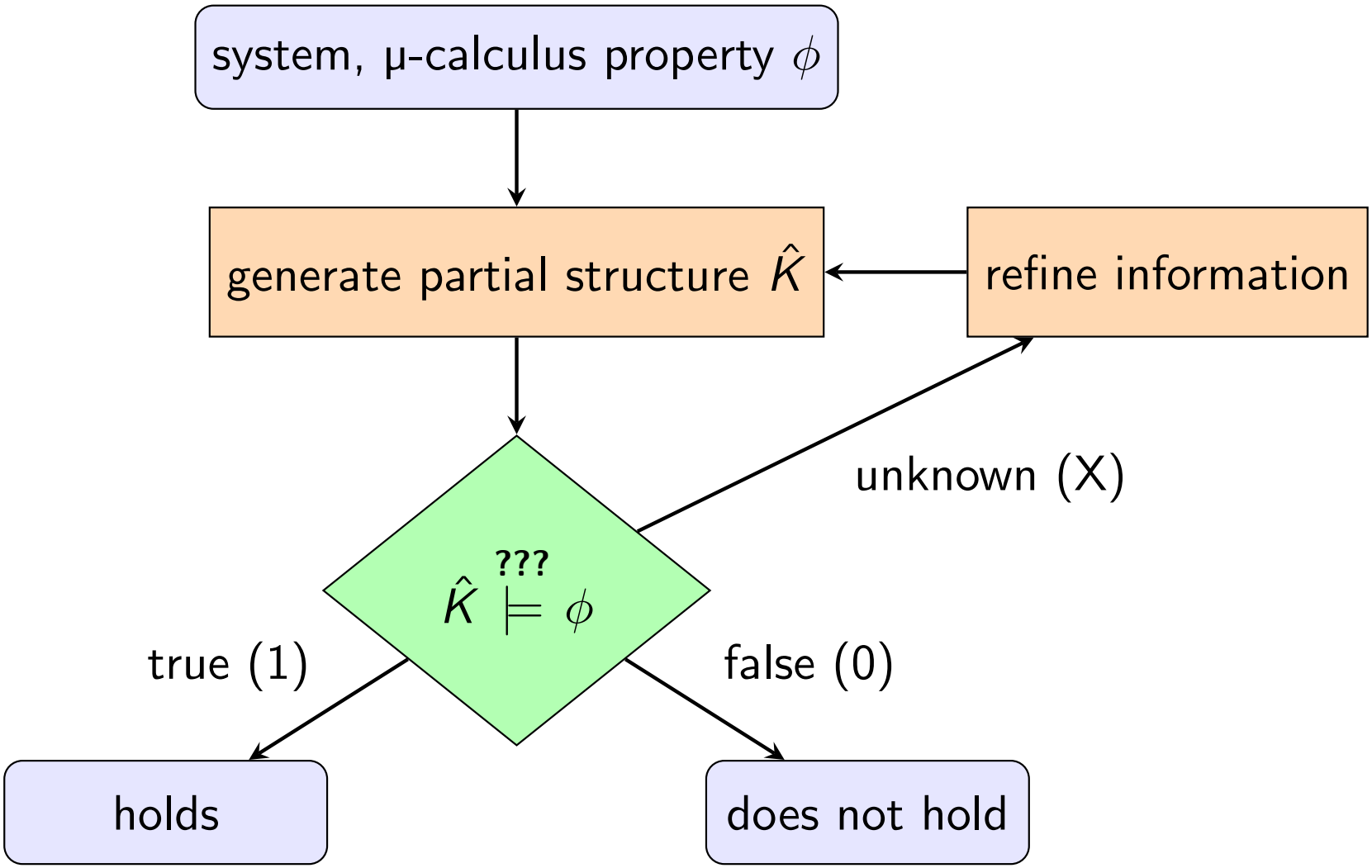
```
fn main() {
    ...
    loop {
        ...
        while (button_not_pressed()) {}
        label: do_something();
        ...
    }
}
```

- **AG**[**EF**[`label`]]: From every state, we can eventually reach `label` with some sequence of inputs ("recovery" / no loss of capability)
  - Computation Tree Logic (CTL) property, not path-universal
- Simple path-universal **AF**[`label`], **AGF**[`label`], ... not helpful here
  - Do not hold even without bug (can run without button ever pressed)
- **Useful properties not verifiable by CEGAR**
- Input-based Three-valued Abstraction Refinement: using a recovery property, we were able to to find a bug in a real-life program

Counterexample-guided Abstraction Refinement (CEGAR)



Three-valued Abstraction Refinement (TVAR)



TVAR frameworks and tools

- Various TVAR frameworks introduced 2001–2011
  - Theoretically and practically complicated
- Nowadays: many CEGAR tools, no practically used TVAR tools
- Blind spots in verification and design
  - Why consider a property you cannot verify?
- How can we improve on this?

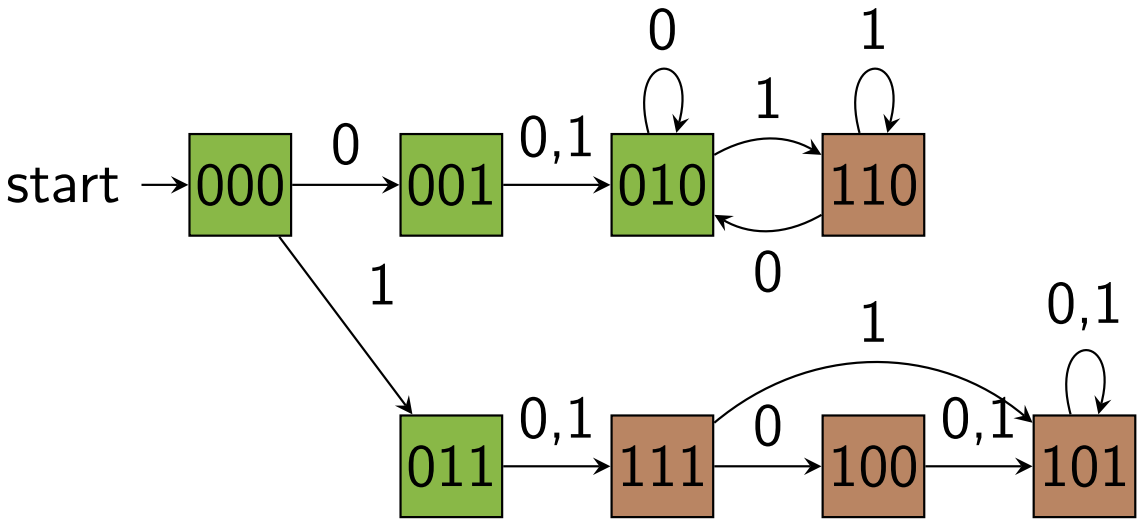
TVAR frameworks and tools

- Various TVAR frameworks introduced 2001–2011
  - Theoretically and practically complicated
- Nowadays: many CEGAR tools, no practically used TVAR tools
- Blind spots in verification and design
  - Why consider a property you cannot verify?
- How can we improve on this?

- Introduced a new TVAR framework simpler than previous ones
- Implemented a tool **machine-check** supporting μ-calculus
- Found a bug in a real-life program using a recovery property

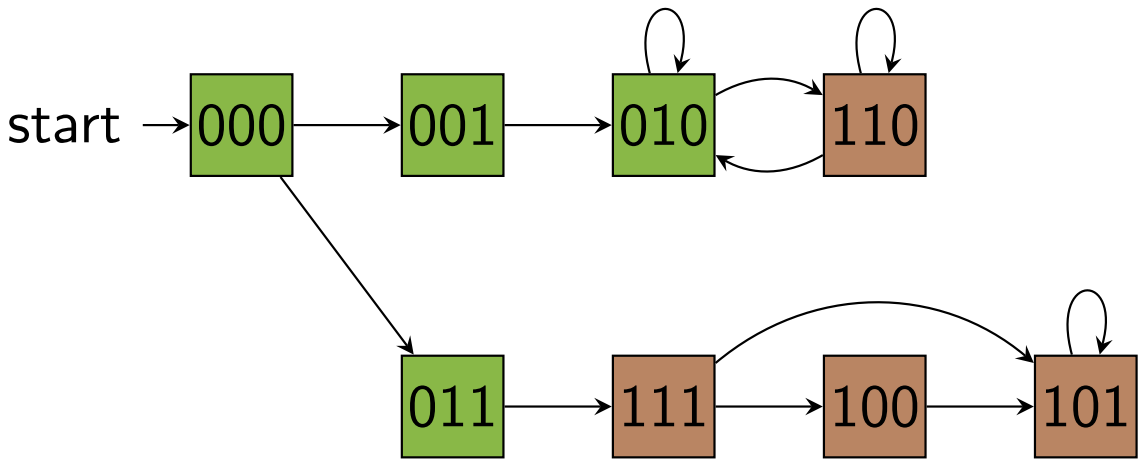
To explain the framework, we need to go deeper...

Example system



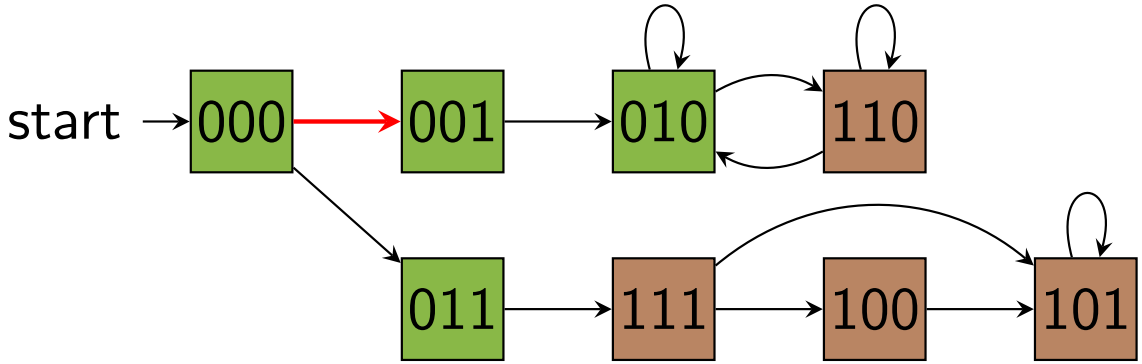
- Automaton with labelling `msb` (most significant bit)
- Property: **AG**[**EF**[`msb` $= 0$]], i.e. we can always recover to `msb` $= 0$
- The system is buggy: if the first input is 1, we will not be able to recover from 111, 100, 101

Classic model checking



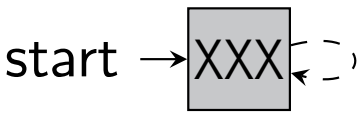
- Turn the automaton into a Kripke structure by forgetting inputs
- Can compute **AG**[**EF**[$\mathtt{msb} = 0$]] does not hold
- . . . but needs to consider all states

CEGAR: abstracting the state space

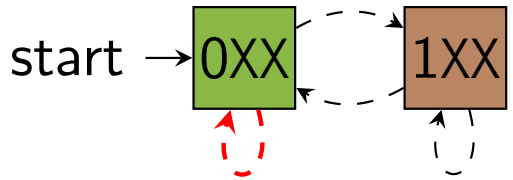


- CEGAR overapproximates
  - states by abstract states (X $\approx$ can be 0 or 1)
  - **transitions by may-transitions** (drawn dashed)
- Cannot prove existence of paths within abstraction!

(a) Before refinement

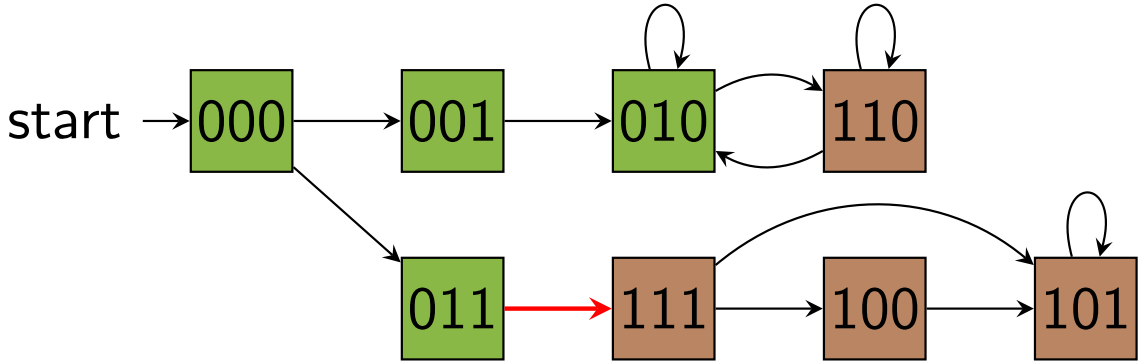


(b) After splitting states



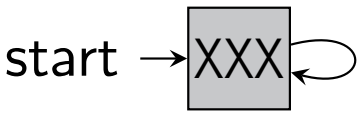
0XX $\rightarrow$ 0XX has to be a may-transition due to e.g. 000 $\rightarrow$ 001

Previous TVAR frameworks

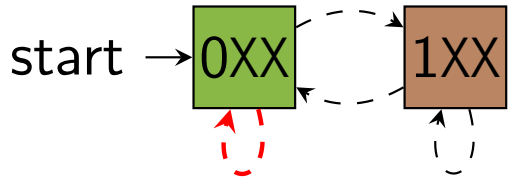


Must-transition (drawn solid): **all** states within an abstract state must have a transition to a state within the successive abstract state

(a) Before refinement



(b) After splitting states



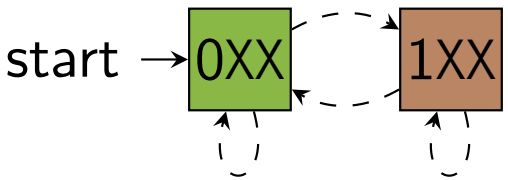
0XX $\rightarrow$ 0XX cannot be a must-transition due to 011 $\rightarrow$ 111

Example: can no longer prove any successor exists after refinement

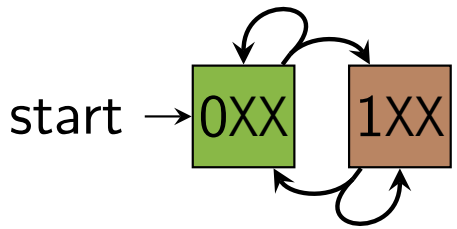
Previous TVAR frameworks: monotonicity woes

- Frameworks with may-transitions + must-transitions can lose provable properties after refinement (*non-monotonicity*)
- Monotone frameworks: more complicated structures

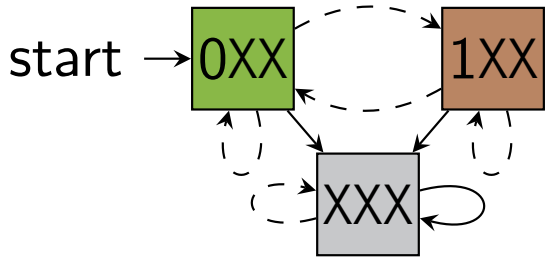
Kripke Modal Transition Structure (KMTS)



Generalised KMTS

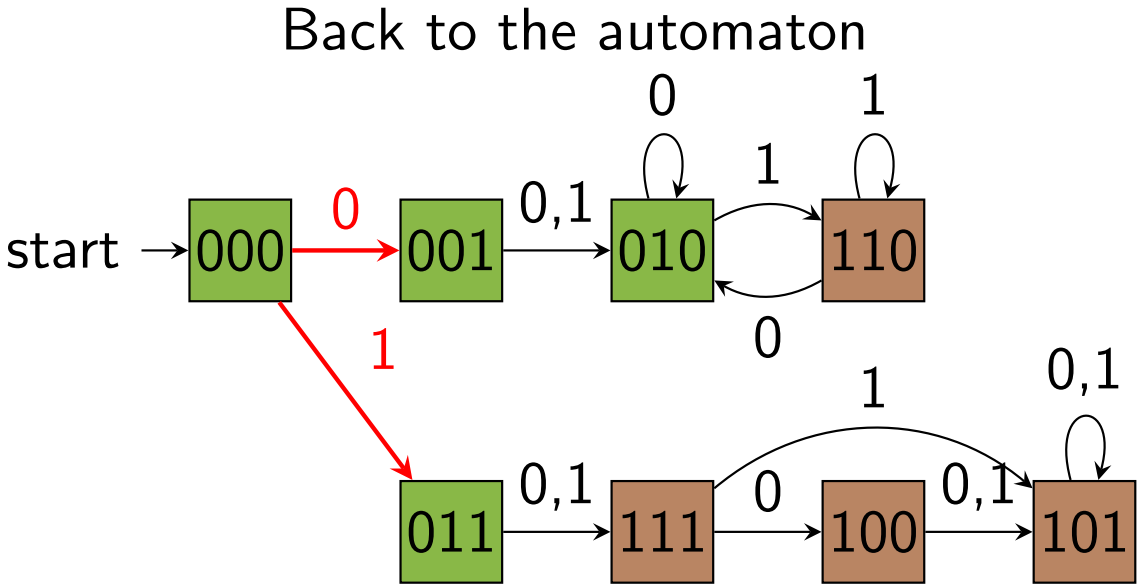


Mixed TS



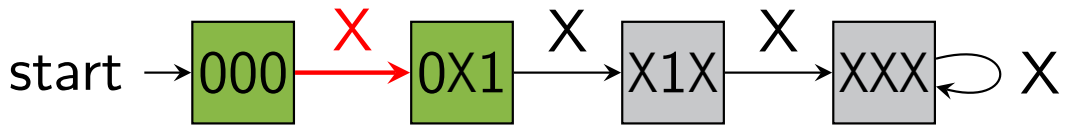
This is necessary due to splitting states.

Input-based TVAR 1/4: abstract simulation



Use *abstract simulation* to compute successive states

Initially with all inputs unknown:

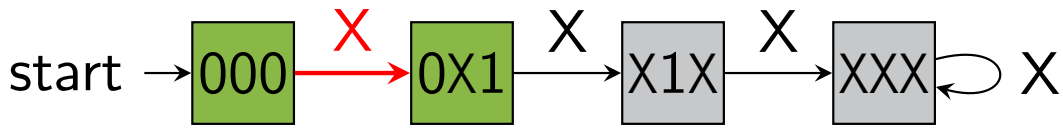


No may/must transitions needed

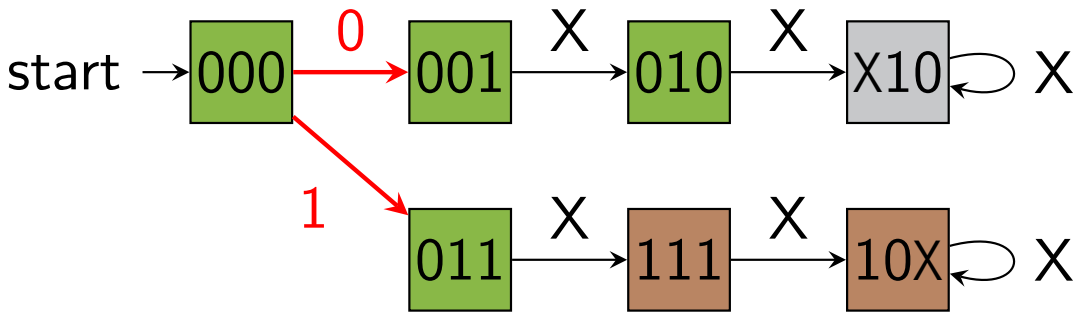
Input-based TVAR 2/4: input splitting

Refinement: Split **inputs** instead of states

Before refinement:



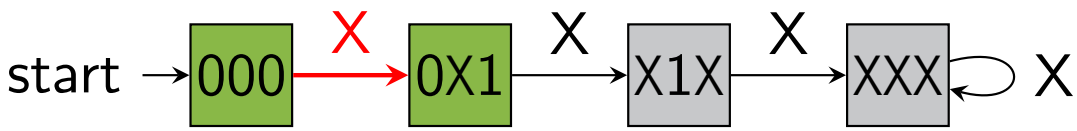
After splitting input from 000,
AG[**EF**[msb = 0]] does not hold:



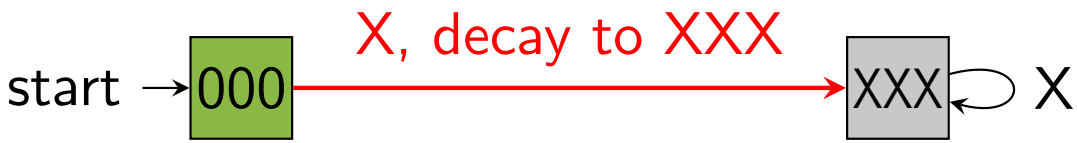
No may/must transitions needed
Partial Kripke Structure, no monotonicity woes

Input-based TVAR 3/4: state decay

New problem: abstract state space unnecessarily big
to compute e.g. `msb = 0` in initial state



Solution: allow *decaying* abstract states
to overapproximate more during abstract simulation



Input-based TVAR 4/4: putting it together

- Refinement by input splitting and lessening decay
- States: existential abstraction
  - Can use Abstract Interpretation to compute abstract simulation
- Proven sufficient requirements for
  - soundness (if we get a result, it is correct)
  - monotonicity (provable properties stay provable)
  - completeness (we get a result in finite time)
- Good choice of abstract domains and refinements crucial

Instantiating tool: **machine-check**

- **Automatic verification of digital systems**
  - System descriptions: strict subset of the Rust language
  - Specification properties: μ-calculus
- Explicit-state model checking
- Three-valued bit-vector abstraction domain + others optional
- Targeted especially to machine-code verification
  - Proofs of concept: AVR and RISC-V machine-code systems

https://machine-check.org

Apache 2.0 / MIT

Experimental evaluation

- 16 machine-code programs for AVR ATmega328P verified
- Using a recovery property, found a bug in a program for Voltage-Controlled Oscillator calibration
  - bug in calibration based on binary search
  - sneaky: loss of calibration accuracy without being obvious

Property	Holds	Refin.	States	Transitions	CPU t. [s]	Mem. [MB]
Inherent (**AG**[w])	✓	513	13059	13573	17.09	87.39
Recovery (**AF**[**AG**[w]])	✗	513	13059	13573	**19.88**	**111.34**
AFG[w] as $\mu X\,.\,\nu Y\,.\,[X] \vee (w \wedge [Y])$ [1]	✓	513	13059	13573	672.05	160.58
AFG[w] as $\mu X\,.\,\nu Y\,.\,[X] \vee (w \wedge [Y])$ [1]*	✓	513	13059	13573	17.98	115.34
Even non-starts ($\nu X\,.\,w \wedge [\,[X]\,]$) [2]	✗	0	17	18	<0.01	3.78
Stack above 0x08FD (**AG**[w])	✓	513	13059	13573	17.08	87.30
Stack above 0x08FE (**AG**[w])	✗	0	17	18	<0.01	3.94

[1] In μ-calculus, LTL, and CTL*, not in CTL
[2] In μ-calculus, not in CTL*
* Guiding abstract state space generation by inherent property

Current & future work

- Ongoing work on **machine-check**
  - E.g. new RISC-V system implementation (December)
- Relative simplicity of framework invites extensions
  - Parametric systems implemented in **machine-check**
- Transfer of ideas between SAT/SMT/QBF solving and model checking
  - CEGAR tools can solve SAT problems
  - TVAR tools can solve QBF problems
  - Parallels between abstraction refinement and DPLL solvers

Conclusion

- New abstraction-refinement framework for μ-calculus verification
  - Splitting using inputs: no problems with transitions
- Works in practice, as evidenced by **machine-check**
  - More work needs to be done for industrial usability
- Verification benchmarks and competitions currently focus on path-universal properties
  - Now we can do more, as we can have the tools!