Cosa2:

A Performant, Adaptable, and Extensible SMT-based Model Checker

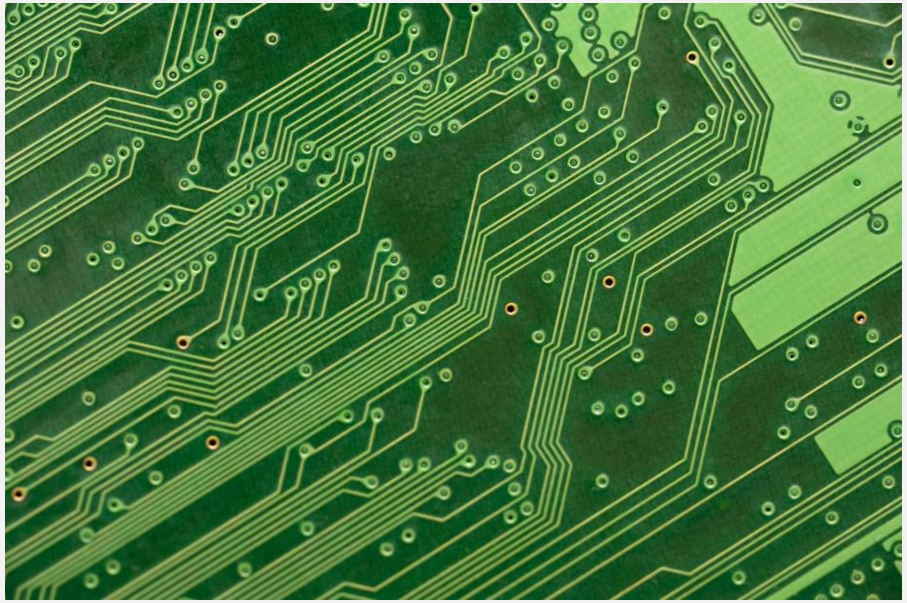
Core Developers: Makai Mann, Ahmed Irfan, Florian Lonsing, Clark Barrett
Contributors: Yahan Yang, Mathias Preiner, Aina Niemetz

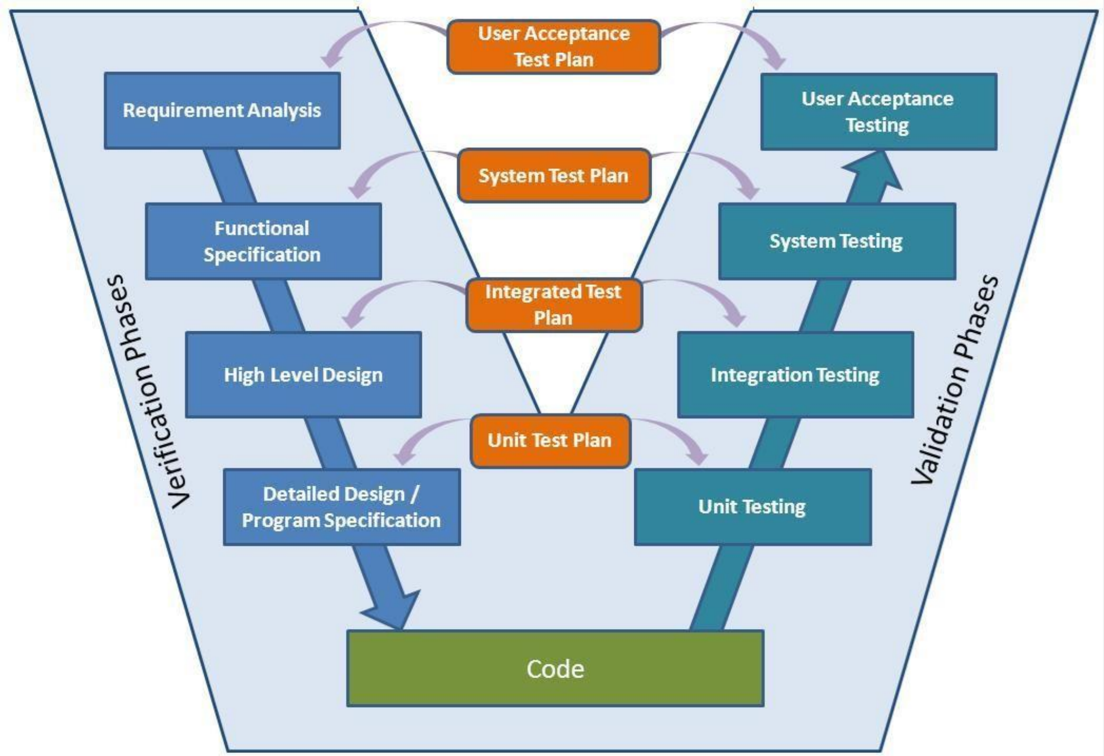
Outline

- Role of Formal Verification
- Model Checking and Cosa2
- Primary Project Goals: The MiniSAT for model checking
- Cosa2 Interface
- [Time Permitting] Inductive Proofs and Example

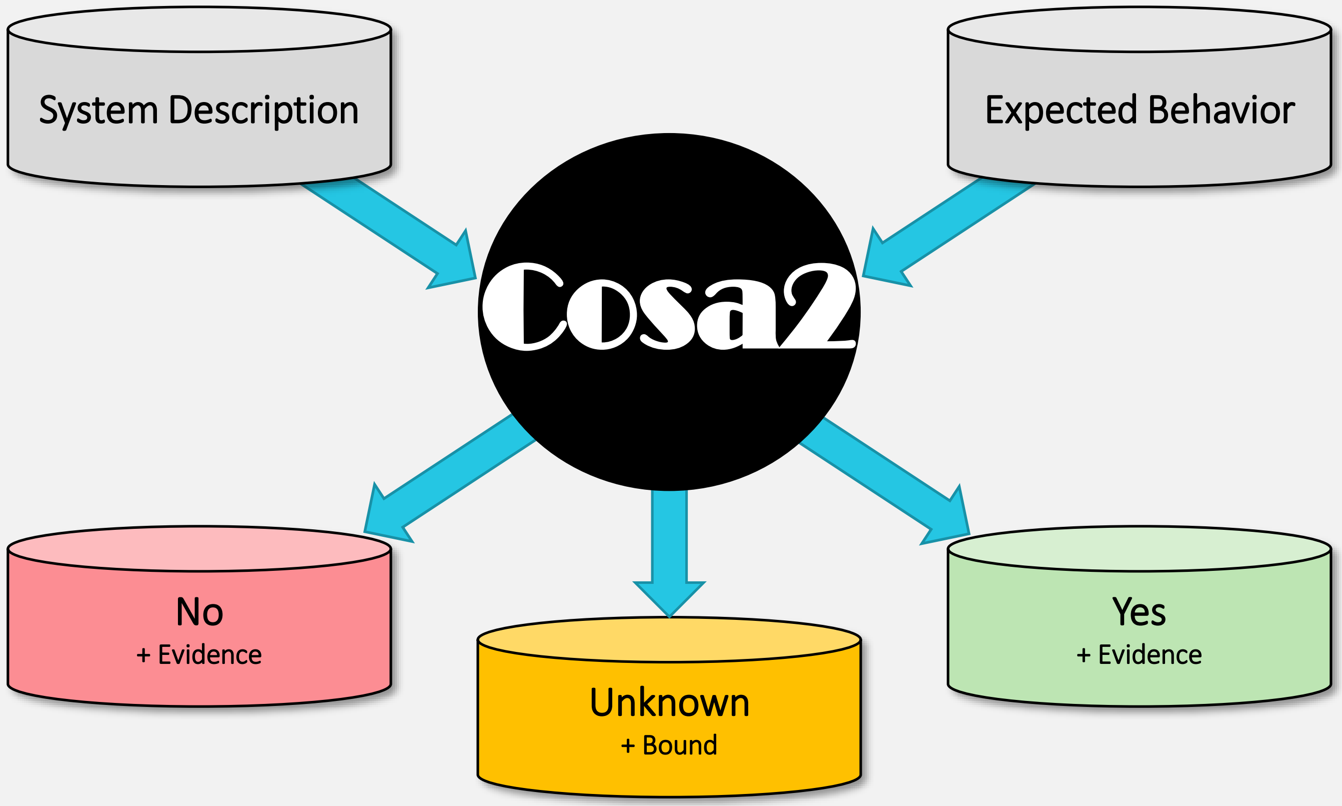
Role of Formal Verification

- Exhaustive Coverage
  - To avoid expensive or threatening failures
- Goal: Left side of V as automated & foolproof as possible
  - Make right side of V faster / less necessary





Model Checking



- Given a **System Description** (M) and an **Expected Behavior** (φ), check if $M \vDash \varphi$
- **Bounded** techniques can only provide **No** or **Unknown (bounded proof)**
- **Unbounded** techniques can also prove the property: answer **Yes**

Project Goal: “The MiniSAT of Model Checking”

- Performant: Competitive Implementations of Standard Model Checking Algorithms
- Adaptable: Clean Interface
- Extensible: Infrastructure for Prototyping New Algorithms

Performant: Competitive Implementations of Algorithms

- Existing Algorithms
  - Bounded Model Checking
  - Bounded Model Checking with Simple Path Check
  - K-Induction
  - Interpolant-based Model Checking

- Has all the same *safety checking* algorithms as CoSA1

- Up Next
  - Various forms of IC3/PDR: Functional IC3, Syntax-guided IC3, IC3 via Implicit Predicate Abstraction
  - Practical Optimizations: Cone of influence reduction
  - CEGAR techniques: Lazy functional abstraction, ProphIC3

Performant: Hardware Model Checking Competition 2019

sponsors

Oski TECHNOLOGY

JKU

FWF
Der Wissenschaftsfonds.

Results

In the SINGLE bit-vector track the top three places are:

1. **AVR**
   Aman Goel, Karem Sakallah (University of Michigan)
2. **CoSA2**
   Makai Mann, Ahmed Irfan, Florian Lonsing, Clark Barrett (Stanford University)
3. **CoNPS-btormc-THP**
   Norbert Manthey (hobbyist, former postdoc @ TU Dresden)

In the SINGLE bit-vector+array track the top three places are:

1. **CoSA2**
   Makai Mann, Ahmed Irfan, Florian Lonsing, Clark Barrett (Stanford University)
2. **AVR**
   Aman Goel, Karem Sakallah (University of Michigan)
3. **CoNPS-btormc-THP**
   Norbert Manthey (hobbyist, former postdoc @ TU Dresden)

Oski Award

CoSA2 for solving the largest number of benchmarks overall.

Performant: Comparison to CoSA

- Case Study with large Transition System
  - SQED on Black Parrot RISC-V core developed at UW
  - Large design – lots of error checking
  - Cosa2 – reaches bound 10 in 14s
  - CoSA with same solver – runs out of memory at bound 10 after 3m38s

Performant: Comparison to CoSA

- Case Study with large Transition System
  - SQED on Black Parrot RISC-V core developed at UW
  - Large design – lots of error checking
  - Cosa2 – reaches bound 10 in 14s
  - CoSA with same solver – runs out of memory at bound 10 after 3m38s

- Run on HWMCC19 (smaller benchmarks) with TO=1hr, Mem=8gb, maximum bound = 200
  - Portfolios: CoSA1 - 317/688, Cosa2 - 326/688
  - Breakdown by engine tells different story
    - BMC/K-induction about the same
    - Interpolant-based: CoSA1 - 83, Cosa2 – 157
    - Many of the ones CoSA1 failed on were picked up by BMC or K-Induction eventually

Performant: Comparison to CoSA

- Conclusion: expected results
  - Cases where Cosa2 is much better, *lots* of cases where it doesn’t matter
  - Reason to believe difference will increase for algorithms that do more work outside of solver
    - Interpolants need to be translated to PySMT level in CoSA1 (slow for “bad” i.e. complex interpolants)
    - Algorithms like IC3 or CEGAR techniques do more work outside the SMT solver

Adaptable: Limitations of a Black Box

- What matters in model checking? Invariant Property Checking!!

- Issues
  - The way a problem is encoded to invariant checking can have drastic performance impacts
  - Straightforward invariant checking not a big portion of the problems in AHA
  - Common complaint from users of commercial formal tools is lack of insight
    - Get yes/no/non-terminating
    - Not even clear what algorithm is used
    - Custom proofs are difficult (not impossible) to construct
    - Tweaks to search strategy are usually impossible

Adaptable: Common Problems in AHA

- **Equivalence Checking**
  - **Many important problems are stateful equivalence checking**
- Lake memory access mapping
- Debug infrastructure
  - Coverage analysis
  - Fault injection?
- PEak mapping with state

Adaptable: Integrated Verification

- All these applications benefit from integrated verification
  - incorporate model checking earlier
  - additional structure is important
- This is **not** a novel idea
  - But it *is* hard in practice
- Moving target for collateral and design tools
  - New design tools
  - Lots of different applications
  - Slow feedback loops

CoSA: Integrated Verification for Agile Hardware Design

Cristian Mattarei, Makai Mann, Clark Barrett, Ross G. Daly, Dillon Huff, and Pat Hanrahan
Stanford University
Stanford, California (USA)
{mattarei, makaim, clarkbarrett, ross.daly, dhuff, pmh}@stanford.edu

***Abstract*—Symbolic model-checking is a well-established technique used in hardware design to assess, and formally verify, functional correctness. However, most modern model-checkers encode the problem into propositional satisfiability (SAT) and do not leverage any additional information beyond the input design, which is typically provided in a hardware description language such as Verilog.**

In this paper, we present CoSA (CoreIR Symbolic Analyzer), a model-checking tool for CoreIR designs. CoreIR is a new intermediate representation for hardware. CoSA encodes model-checking queries into first-order formulas that can be solved by Satisfiability Modulo Theories (SMT) solvers. In particular, it natively supports encodings using the theories of bitvectors and arrays. CoSA is closely integrated with CoreIR and can thus leverage CoreIR-generated metadata in addition to user-provided lemmas to assist with formal verification. CoSA supports multiple input formats and provides a broad set of analyses including equivalence checking and safety and liveness verification. CoSA is open-source and written in Python, making it easily extendable.

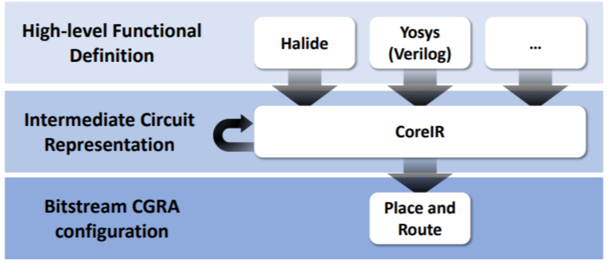
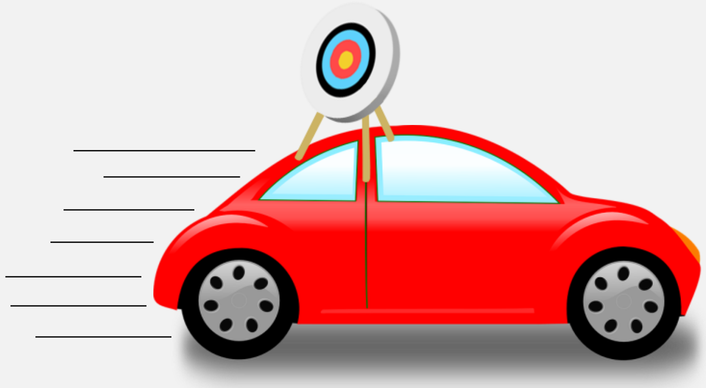


Fig. 1. AHA Flow

CoSA was developed as a tool for verifying correctness at various stages of the toolflow in the Agile Hardware (AHA) Project at Stanford University [18]. This project aims to improve performance and design productivity by incorporating ideas from agile software development to speed up the development cycle.



Adaptable: Big Picture on Clean Interface

- Need more than a black box model checker
  - Clean API for encoding specific problems
  - Transparent suite of algorithms for trying on different problems
  - Different backend solvers to rely on improvements and new techniques

- Clarifications
  - **Not** advocating that designers become experts in formal methods
  - **Not** removing support for push-button techniques

- Goal
  - Allow interested users to encode their own unique problems
    - Should understand transition systems but not necessarily model checking algorithms
  - Straightforward interface for communicating with model checker

Extensible: Infrastructure for Prototyping New Algorithms

- Adaptability is for users
- Extensibility is for developers

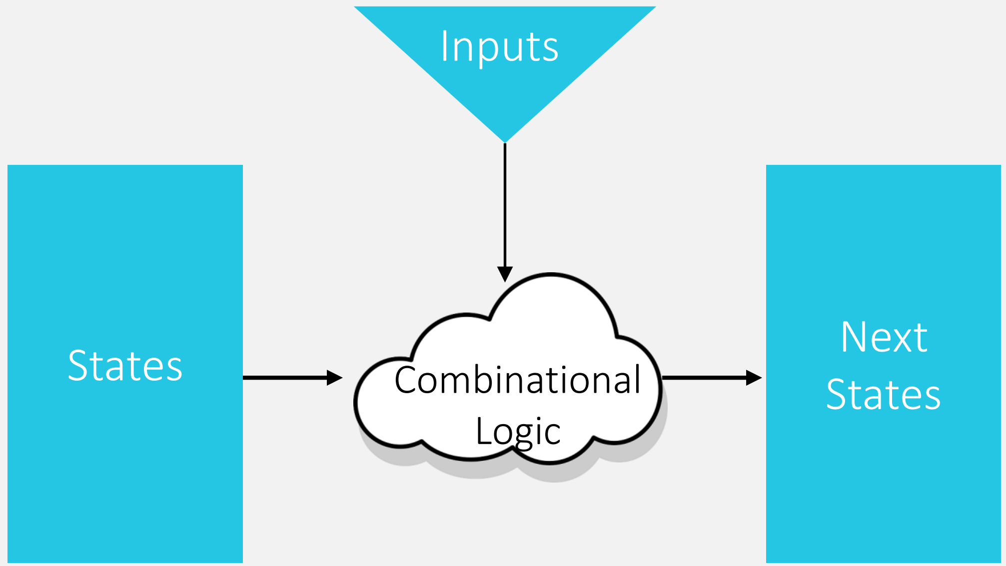
cheating: removed comment here

```cpp
Bmc::Bmc(const Property & p, SmtSolver & solver) : super(p, solver)
{
  initialize();
}

Bmc::~Bmc() {}

void Bmc::initialize()
{
  super::initialize();
  solver_->assert_formula(unroller_.at_time(ts_.init(), 0));
}

ProverResult Bmc::check_until(int k)
{
  for (int i = 0; i <= k; ++i) {
    if (!step(i)) {
      return ProverResult::FALSE;
    }
  }
  return ProverResult::UNKNOWN;
}

bool Bmc::step(int i)
{
  if (i <= reached_k_) {
    return true;
  }

  bool res = true;
  if (i > 0) {
    solver_->assert_formula(unroller_.at_time(ts_.trans(), i - 1));
  }

  solver_->push();
  logger.log(1, "Checking bmc at bound: {}", i);
  solver_->assert_formula(unroller_.at_time(bad_, i));
  Result r = solver_->check_sat();
  if (r.is_sat()) {
    res = false;
  } else {
    solver_->pop();
  }

  ++reached_k_;

  return res;
}
```

- Goal: Provide expressive, easily understood infrastructure for model checking algorithms
  - Allow formal methods experts to quickly try new ideas
  - Open-source and **simple** (BMC fits on one screen)

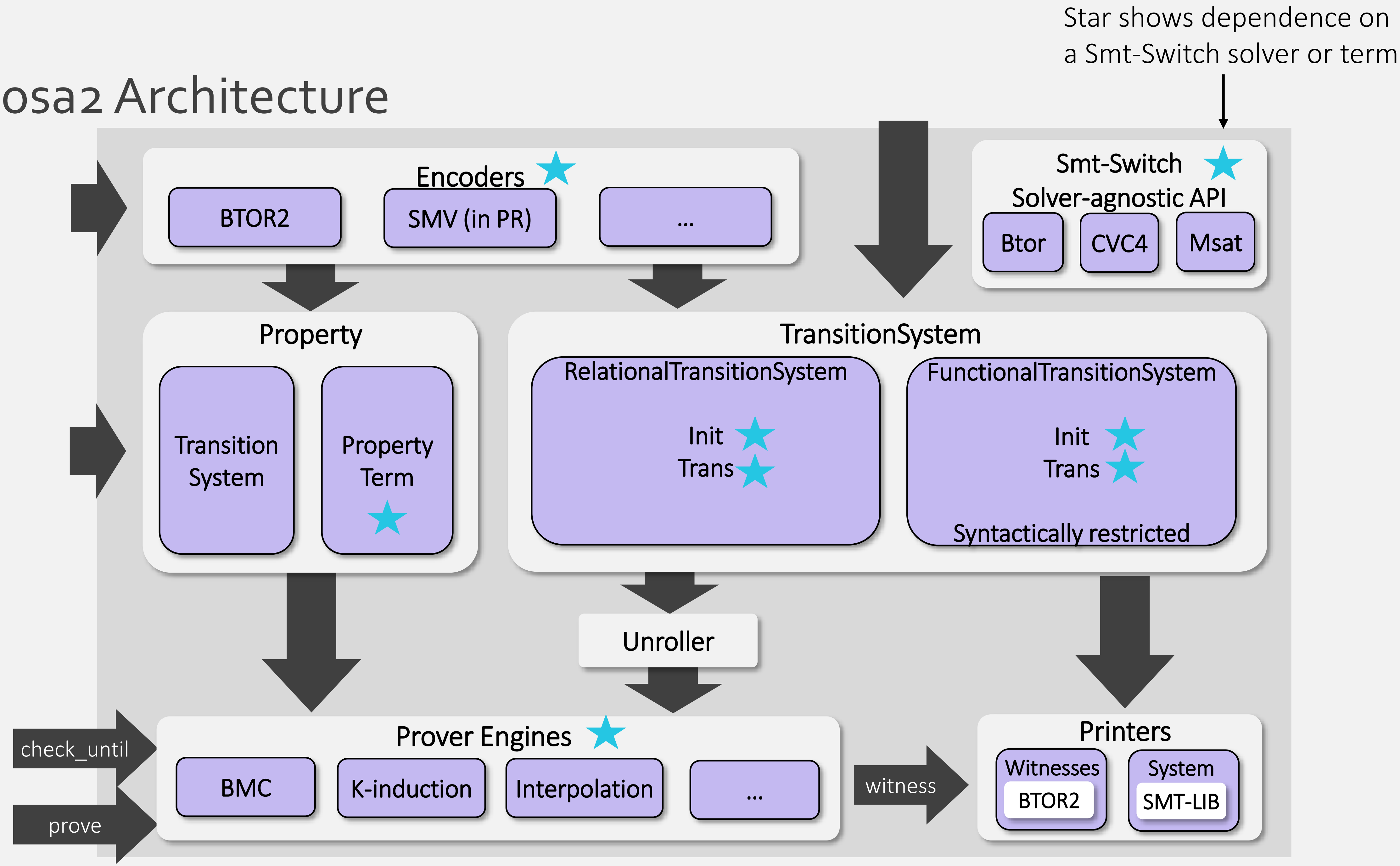
Cosa2: Interface Choices

- Very modular: pieces exposed for interested users
  - If they don't care, can use them as black boxes
  - Use abstract classes with different implementations
- New Python API (basically done – needs more testing)
- Driving principle is *correctness*, occasionally had to sacrifice convenience
  - Don't want to let users shoot themselves in the foot
  - Enforces strict semantics of transition system – get expected behavior
  - Example: restrictions on use of states vs inputs
  - Example: safety properties only over state elements
- Let's look at an architecture diagram



Functional Transition System looks just like hardware

Cosa2 Architecture



Cosa2 TransitionSystem Interface Preview

```cpp
/* Sets initial states to the provided formula
 * @param init the new initial state constraints
 */
void set_init(const smt::Term & init);

/* Add to the initial state constraints
 * @param constraint new constraint on initial states
 */
void constrain_init(const smt::Term & constraint);

/* Set the transition function of a state variable
 *   val is constrained to only use current state variables
 * Represents a functional update
 * @param state the state variable you are updating
 * @param val the value it should get
 */
void assign_next(const smt::Term & state, const smt::Term & val);

/* Add an invariant constraint to the system
 * This is enforced over all time
 * Specifically, it adds the constraint over both current and next variables
 * @param constraint the boolean constraint term to add (should contain only
 * state variables)
 */
void add_invar(const smt::Term & constraint);

/** Add a constraint over inputs
 * @param constraint to add (should not contain any next-state variables)
 */
void constrain_inputs(const smt::Term & constraint);

/** Convenience function for adding a constraint to the system
 * if the constraint only has current states, equivalent to add_invar
 * if there are only current states and inputs, equivalent to
 *   constrain_inputs
 * throws an exception if it has next states (should go in trans)
 */
void add_constraint(const smt::Term & constraint);

/* Gives a term a name
 *   This can be used to track particular values in a witness
 * @param name the (unique) name to give the term
 * @param t the term to name
 *
 * Throws an exception if the name has already been used
 *  Note: giving multiple names to the same term is allowed
 */
void name_term(const std::string name, const smt::Term & t);
```

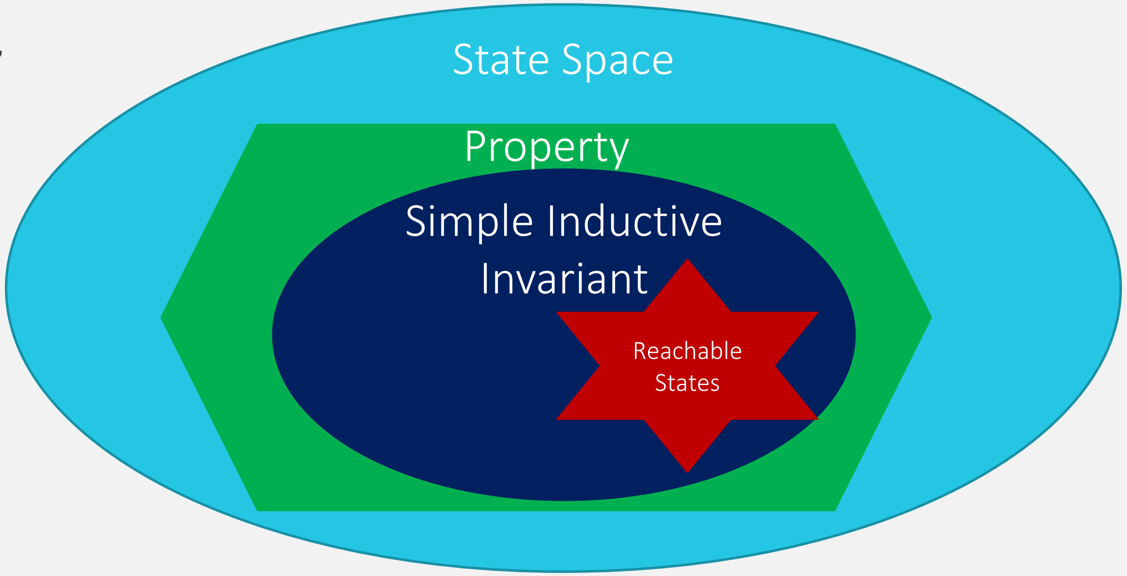
```cpp
/* Create an input of a given sort
 * @param name the name of the input
 * @param sort the sort of the input
 * @return the input term
 */
smt::Term make_input(const std::string name, const smt::Sort & sort);

/* Create an state of a given sort
 * @param name the name of the state
 * @param sort the sort of the state
 * @return the current state variable
 *
 * Can get next state var with next(const smt::Term t)
 */
smt::Term make_state(const std::string name, const smt::Sort & sort);

/* Map all next state variables to current state variables in the term
 * @param t the term to map
 * @return the term with all current state variables
 */
smt::Term curr(const smt::Term & term) const;

/* Map all current state variables to next state variables in the term
 * @param t the term to map
 * @return the term with all next state variables
 */
smt::Term next(const smt::Term & term) const;

/* @param sv the state variable to check
 * @return true if sv is a current state variable
 *
 * Returns false for any other term
 */
bool is_curr_var(const smt::Term & sv) const;

/* @param sv the state variable to check
 * @return true if sv is a next state variable
 *
 * Returns false for any other term
 */
bool is_next_var(const smt::Term & sv) const;
```

Recap of Cosa2 Goals: “MiniSAT of Model Checking”

- Performant: Competitive Implementations of Push-Button Algorithms
- Adaptable: Clean Interface for Adapting to Specific Problems and Providing Hints
- Extensible: Reliable Semantics and Strong Infrastructure for Prototyping New Techniques

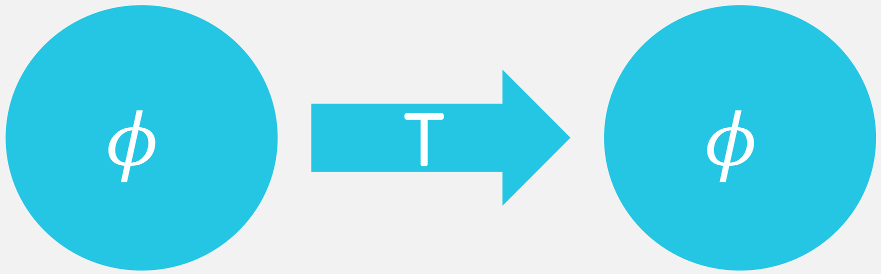
Cracking Open Model Checkers: Inductive Proof

- Big Hammer of Model Checking: Inductive proofs
  - “Property-directed” as opposed to direct reachability
  - K-Induction: strong mathematical induction of property
  - Interpolant-based: refining over-approximations until inductive
  - IC3: educated guessing of possible inductive invariants until convergence
- Difficulty of Transition System / Property pair
  - Hard to quantify for PSPACE problems
  - In practice, appears strongly correlated to “inductiveness”
  - As in programming
    - State is extremely useful
    - State is extremely complex



Inductive Invariant

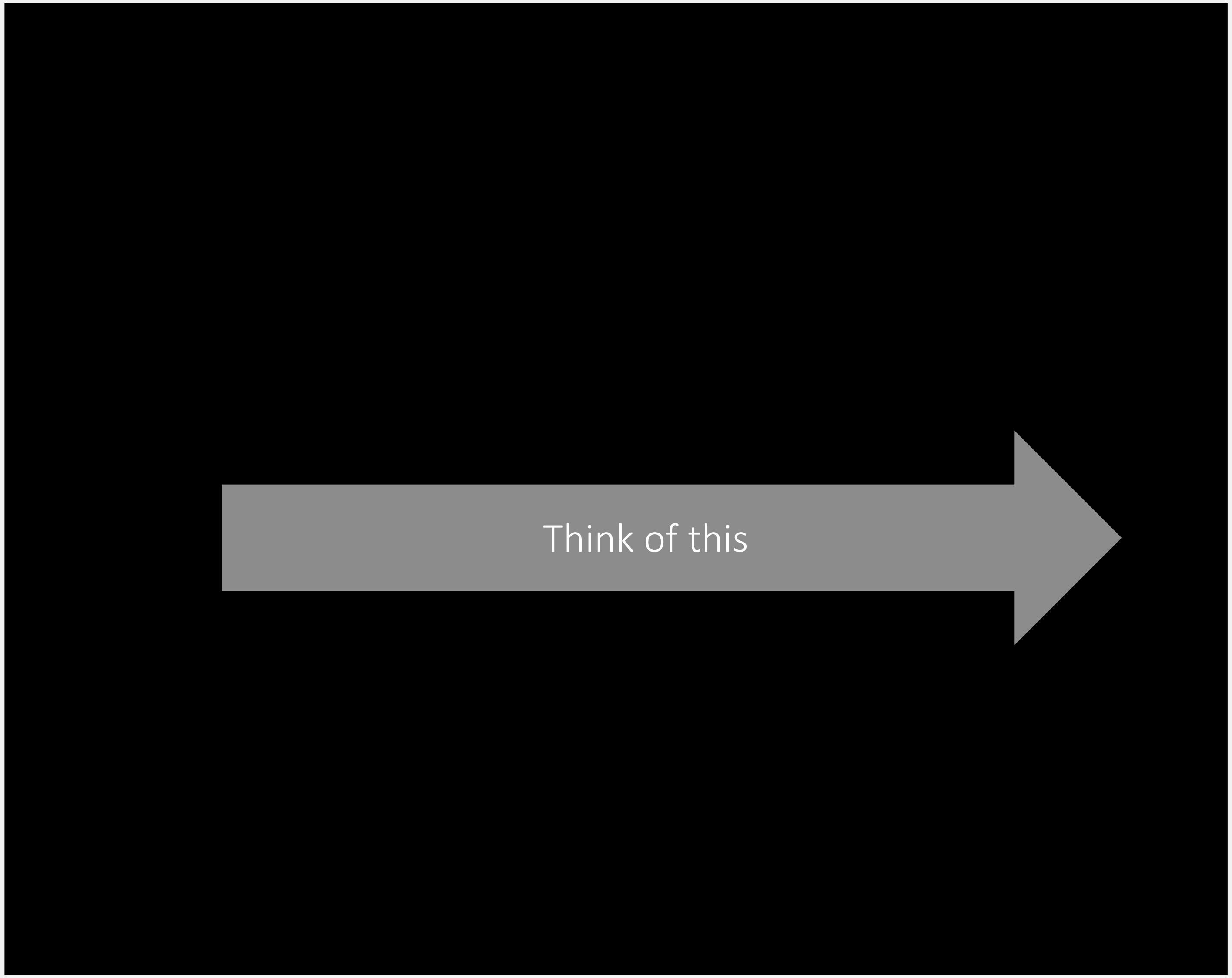
- Symbolic Transition System: $\langle X, I, T \rangle$
  - X : variables, X' are corresponding next state variables
  - $I(X)$: initial state constraints
  - $T(X, X')$: transition relation
- Inductive Invariant: $\phi(X)$
  - $I(X) \vDash \phi(X)$
  - $\phi(X) \wedge T(X, X') \vDash \phi(X')$
- Proving safety property $P(X)$
  - If $\phi(X) \vDash P(X)$
  - Then we've proven the property

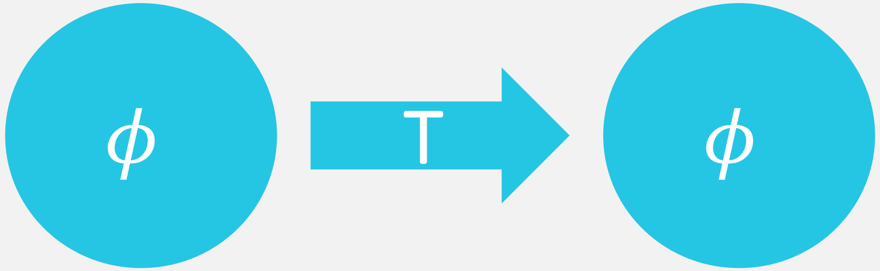


If ϕ holds before a transition, then ϕ *must* hold after a transition

i.e. ϕ describes a *closed* set of states

Inductive Invariant



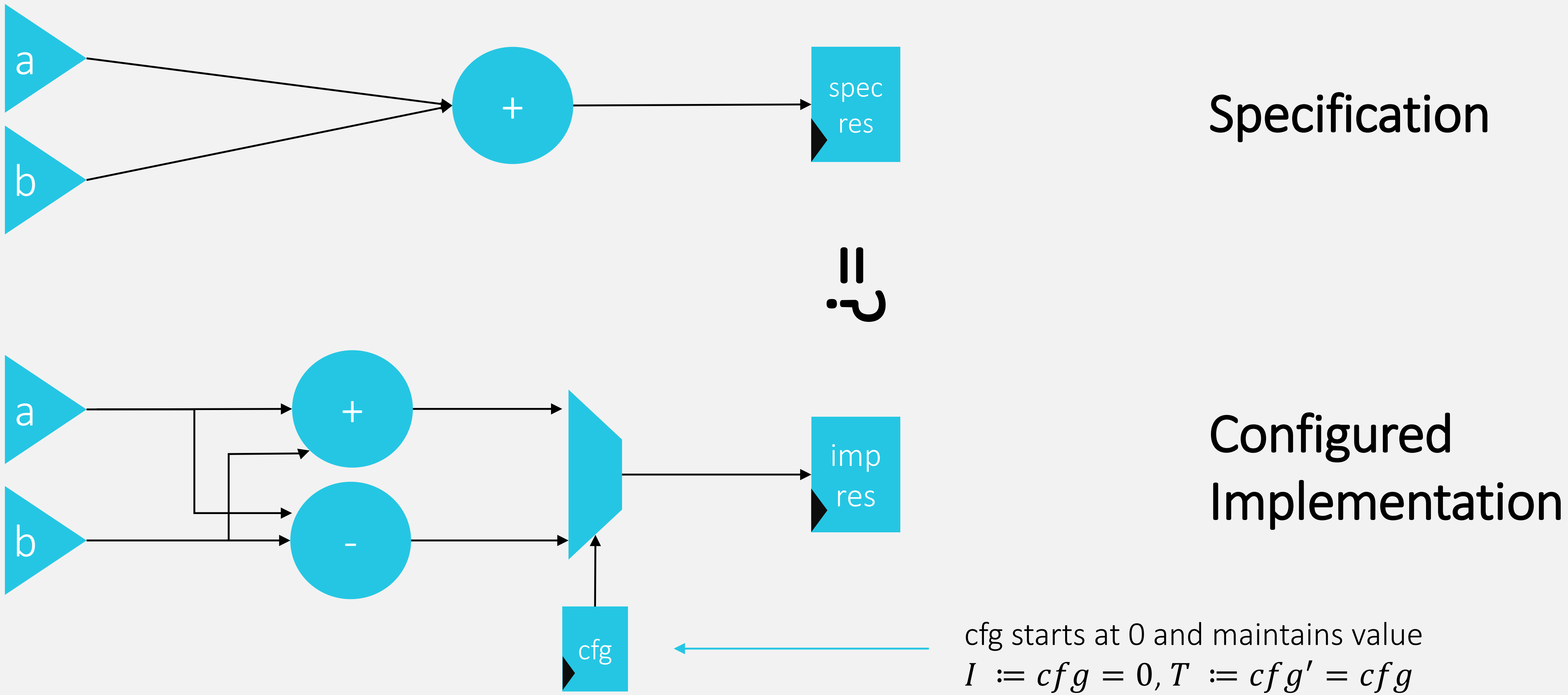


If ϕ holds before a transition, then ϕ *must* hold after a transition

i.e. ϕ describes a *closed* set of states

Example

- Contrived example for demonstrating inductive properties



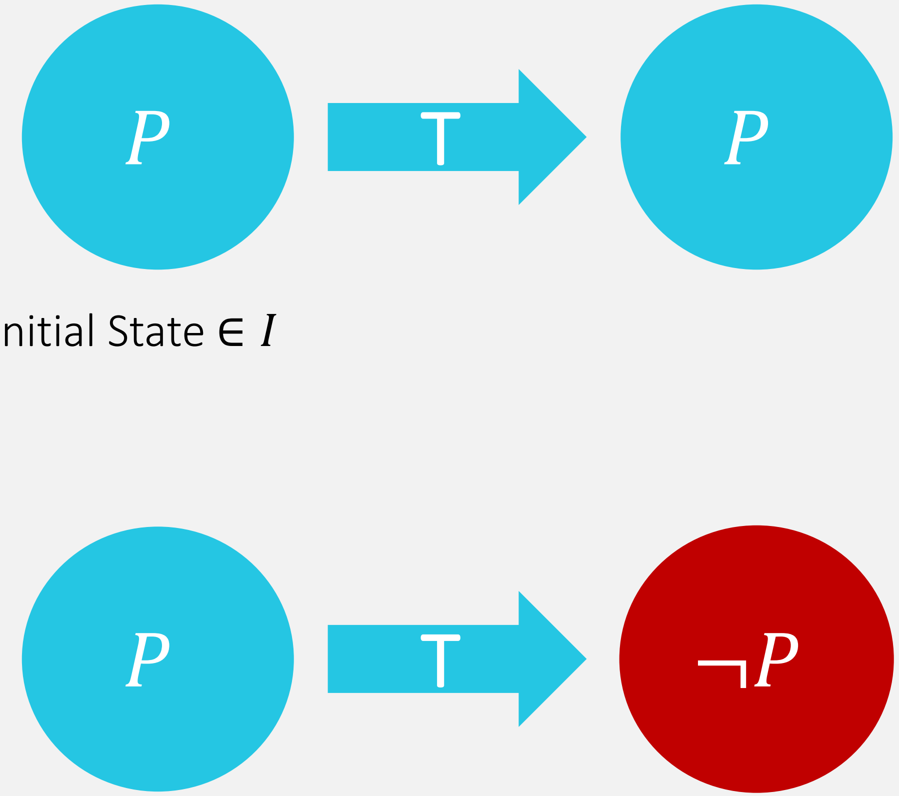
cfg starts at 0 and maintains value
$I \coloneqq cfg = 0, T \coloneqq cfg' = cfg$

Example: Not Inductive

- Property: P $:= \ spec_{res} = imp_{res}$
  - Does not mention cfg at all

- Can start in a P state
  - And transition to a $\neg P$ state

- Alternative *Inductive* Property
  - $P_{ind} \ := cfg = 0 \land spec_{res} = imp_{res}$
  - Manually-strengthened property
  - Not always worth the effort to make fully inductive manually, but the “more inductive”, the better



Base Case

Initial State $\in I$

Inductive Case

Arbitrary P state
$a = 0$
$b = 0$
$spec_{res} = 0$
$imp_{res} = 0$
$cfg = 1$

$a = 2$
$b = 2$
$spec_{res} = 4$
$imp_{res} = 0$
$cfg = 1$

Takeaways

- Cosa2 Goal: MiniSAT of Model Checking
  - Performant
  - Adaptable
  - Extensible
- Simple Transition System Interface
  - Can be manipulated directly for specific problems
- Inductive Proofs
  - Keep induction in mind when using model checkers

Thank You!

- Questions/Comments/Concerns?

- Future Work
  - IC3 Implementations
  - Preprocessing Passes: Cone of Influence, ITE rewriting
  - CEGAR Loops: Functional Abstraction, ProphIC3
  - More frontends: SMV (in a PR by Yahan Yang), CoreIR
  - Temporal logic support
    - Not currently a high priority
    - Lenny working on an alternative solution that should be easier for developers as well

Backup: Comparison to CoSA

Solver	Solved	Unsafe	Safe	Unknown	TO	MO
1-BMC	275	275	0	94	316	3
2-BMC	274	274	0	93	319	2
1-KIND	300	270	30	18	332	38
2-KIND	302	269	33	0	385	1
1-INT	83	54	29	0	524	81
2-INT	157	124	33	2	519	10

Solver	Solved	Unsafe	Safe
CoSA1	317	275	42
Cosa2	326	277	49