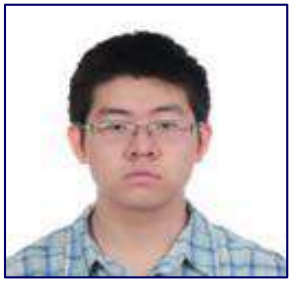
Application 2
Hardware and SoC Verification

Hongce Zhang



William Yang

Grigory Fedyukovich

Sharad Malik

Bo-Yuan Huang

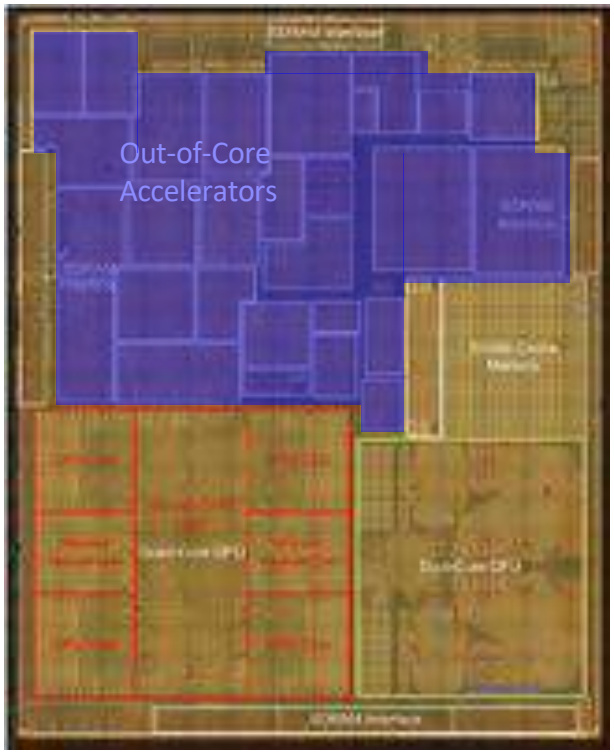
Yue Xing

Pramod Subramanyan

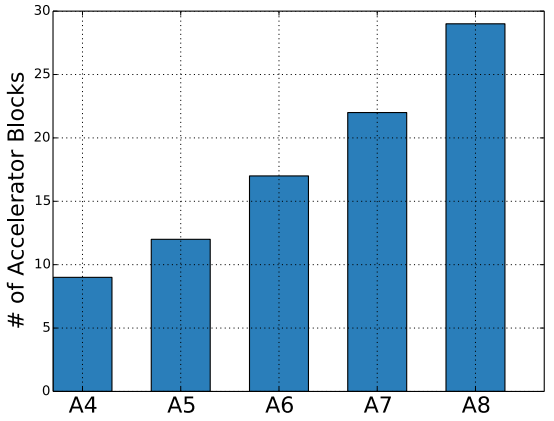
Yakir Vizel

It's a whole new accelerator world…

Apple A8



Software accessible accelerators/peripherals/devices



[www.anandtech.com/show/8562/chipworks-a8]

- Increasing number of accelerators and SW/HW interactions
- Need to verify accelerator implementations
- Need to verify SW with HW interactions

Specifications ?!

Abstractions ?!

Addressing the Specification Gap

Lessons from Processor Instruction Set Architecture (ISA)

HW Abstraction

- Software view of hardware
- Defines semantics for SW interactions with HW
- Enables SW verification

ISA , ILA

ISA, **ILA**

- Specifies *visible state*
  - abstracts details
- *Modular* specification
  - set of instructions

HW Specification

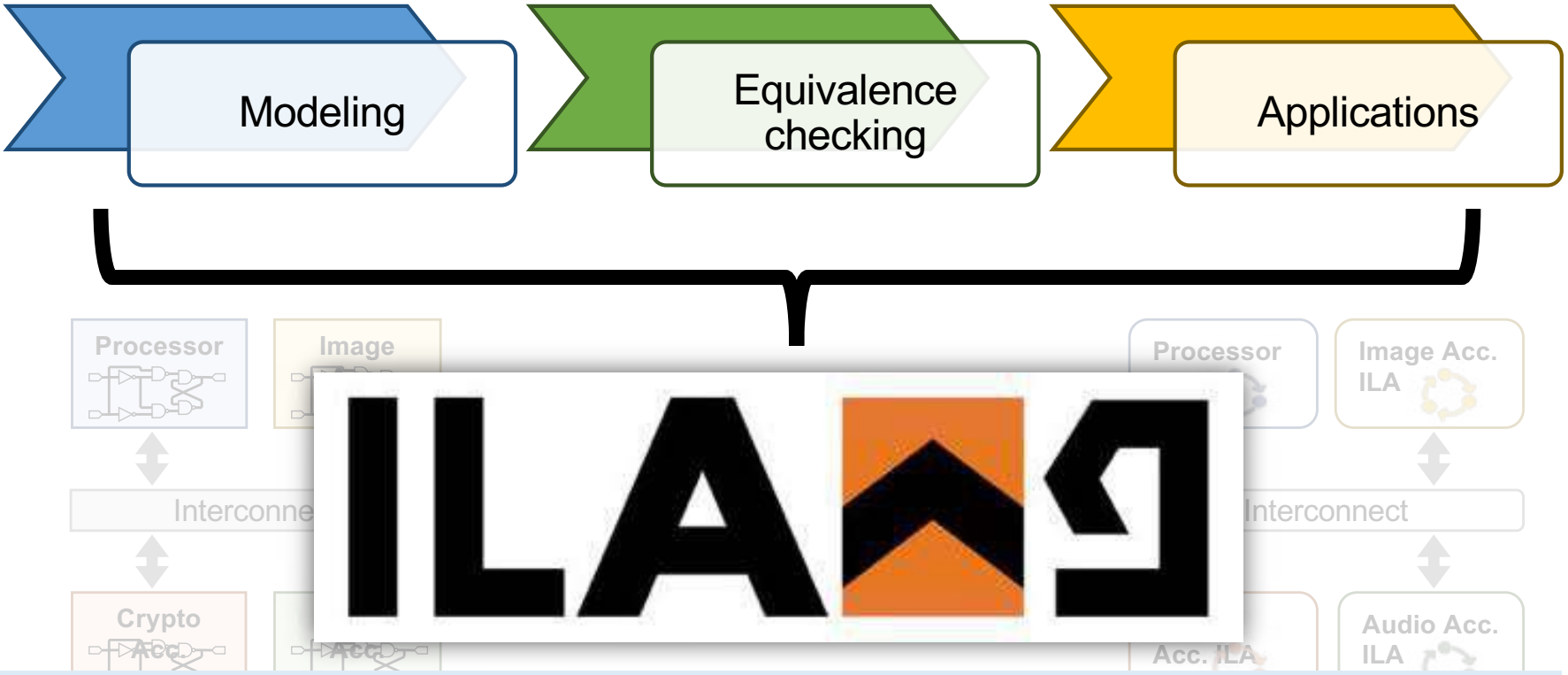
- Specification for HW implementation
- Enables implementation verification
- Enables HW upgrades

ILA: Instruction-Level Abstractions

Huang et al. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. ACM Transactions on Design Automation of Electronic Systems 24(1): 10:1-10:24 (2019)

ILAng Platform for SoC Modeling & Verification

ILA: Instruction-Level Abstractions

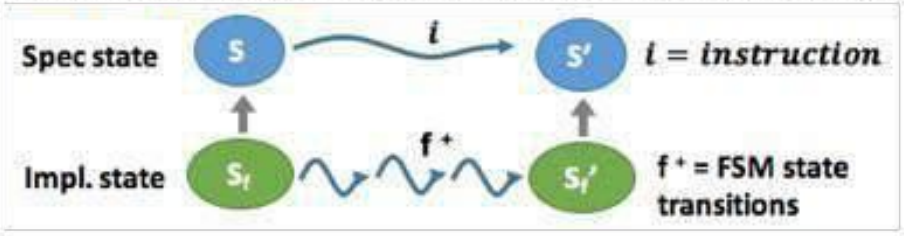


Huang et al. ILAng: A Modeling and Verification Platform for SoCs Using Instruction-Level Abstractions. Proceedings of TACAS Conference (1) 2019: 351-357

https://github.com/Bo-Yuan-Huang/ILAng

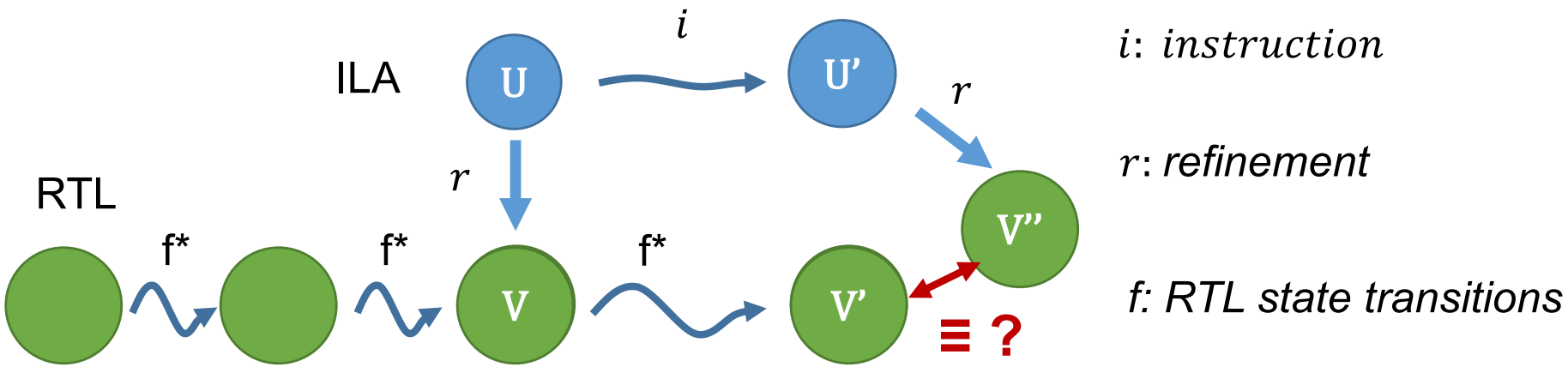
ILA-based Verification

Refinement Relation: what state to match and when to match it



- Per-instruction equivalence checks are "properties"
- Can use model checkers to check these properties

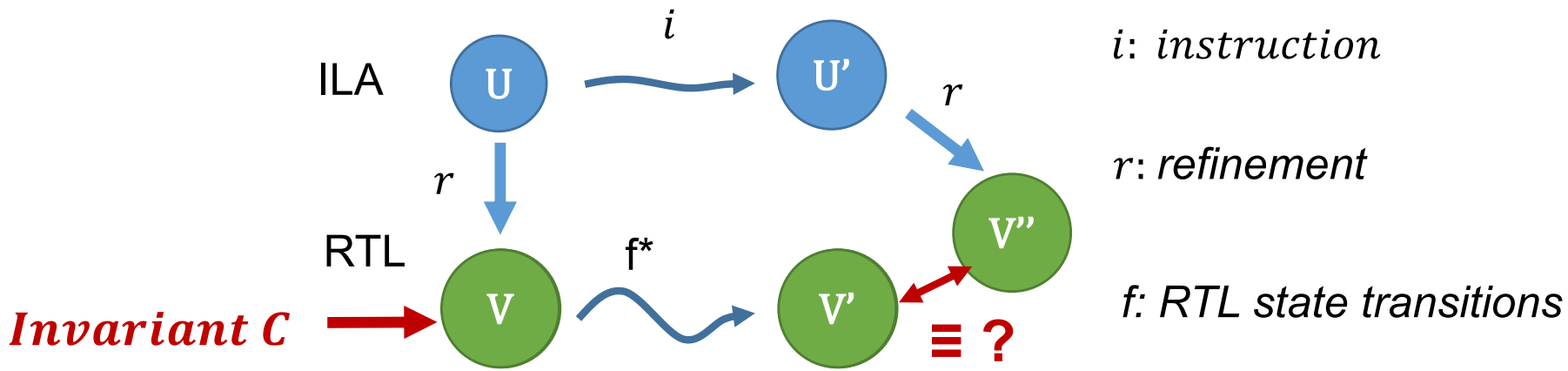
Verification of an Instruction



Check instruction correctness (property):
- in *all* RTL states that the instruction can start in

Verification complexity *explodes* due to:
- arbitrary start state
- large RTL state space (including micro-architecture)

Abstracting the Environment

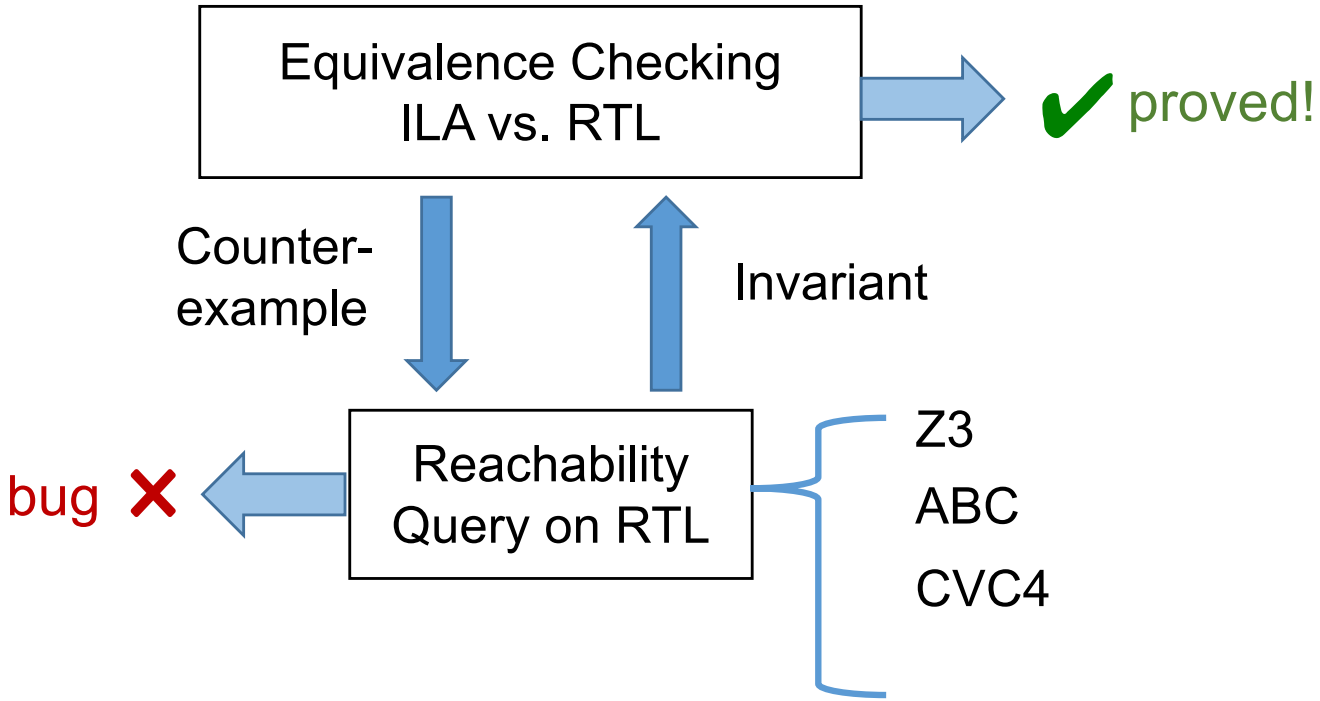


- Known method: abstract the starting state as ***environment invariant***
  - Captures the environment under which a property is checked
  - Often provided *manually* by the designer

- Our goal: ***Automate*** discovery of environment invariants
  - Reduces manual burden on designer
  - Enables scalable model checking for all starting states

Environment Invariant Discovery

- CEGAR loop, with invariant generation



Preliminary Experiments

- 2 synthetic examples: RC (Counters), Simple Pipeline (SP)
- 3 real-world designs: AES, Gaussian Blur are publicly-available accelerators, PicoRV32 is a Risc-V processor

	Monolithic CHC	CEGAR	
Solvers	Z3	ABC	Z3
RC	1.6s	32.3s	16.2s
SP	1035.2s	29.7s	40.4s
AES	T.O.	O.O.M	T.O.
PicoRV32	T.O.	O.O.M	7864.8s
Gaussian Blur	T.O.	T.O.	2783.7s

Computation resources
- Time-limit: 10 hours
- CPU: i5-8300H
- Memory-limit: 32GB
- 1TB SSD

How to do better?

- Monolithic CHC: uses Constrained Horn Clause (CHC) solver in Z3 for finding invariants for the equivalence checking problem

Issues in Invariant Generation

- PDR-based techniques for bit-vectors (e.g., ABC)
  - usually bit-blast the design
  - generate bit-level invariants (word-level ABC did not work well)

  - Example: AES block encryption accelerator
    - Invariant: $Status \neq 0 \rightarrow Cnt_{AES} = Cnt_{blk} + IV$
    - Cnt_{AES}, Cnt_{blk}, IV are all 128-bits
    - Resulted in a giant expression at the bit-level!

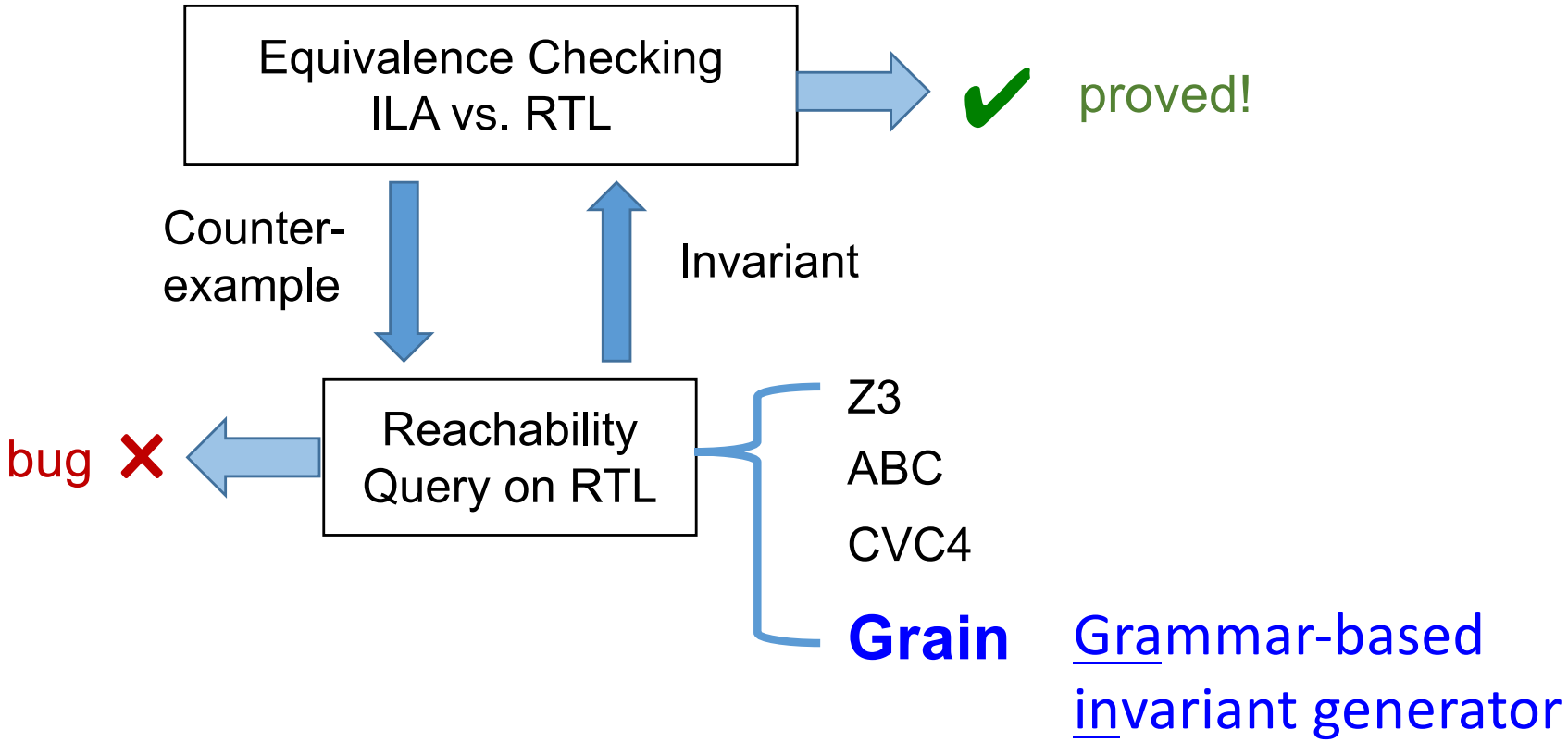
1. work on the word-level

- Invariants seem to get lost
  - usually the invariants have nothing to do with a *concrete value* in the data processing pipeline
  - hard to steer the proof/interpolants in the solver

2. use design insights to guide solver

Environment Invariant Discovery

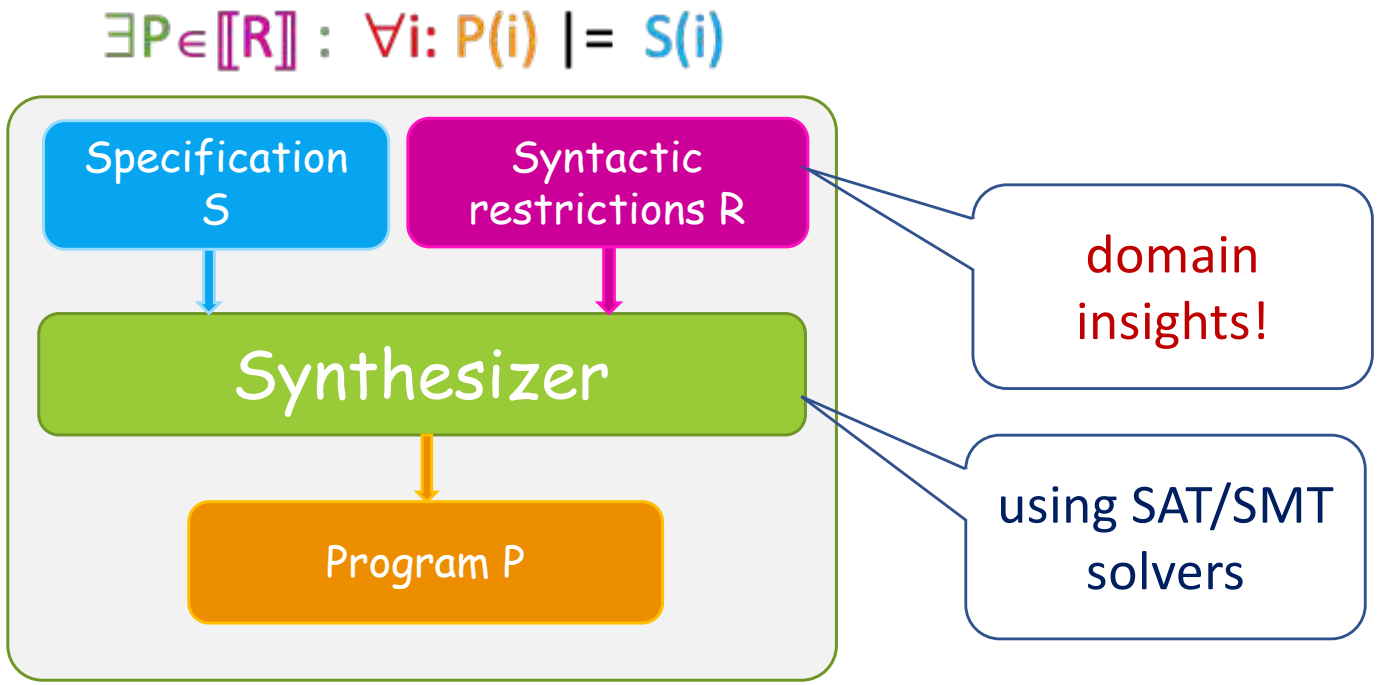
- CEGAR loop, with invariant generation



Zhang et al. **Synthesizing Environment Invariants for Modular Hardware Verification.** Proceedings of VMCAI 2020: 202-225.

Syntax-Guided Synthesis (SyGuS)

[R. Alur et al. Syntax-guided Synthesis. FMCAD'13]



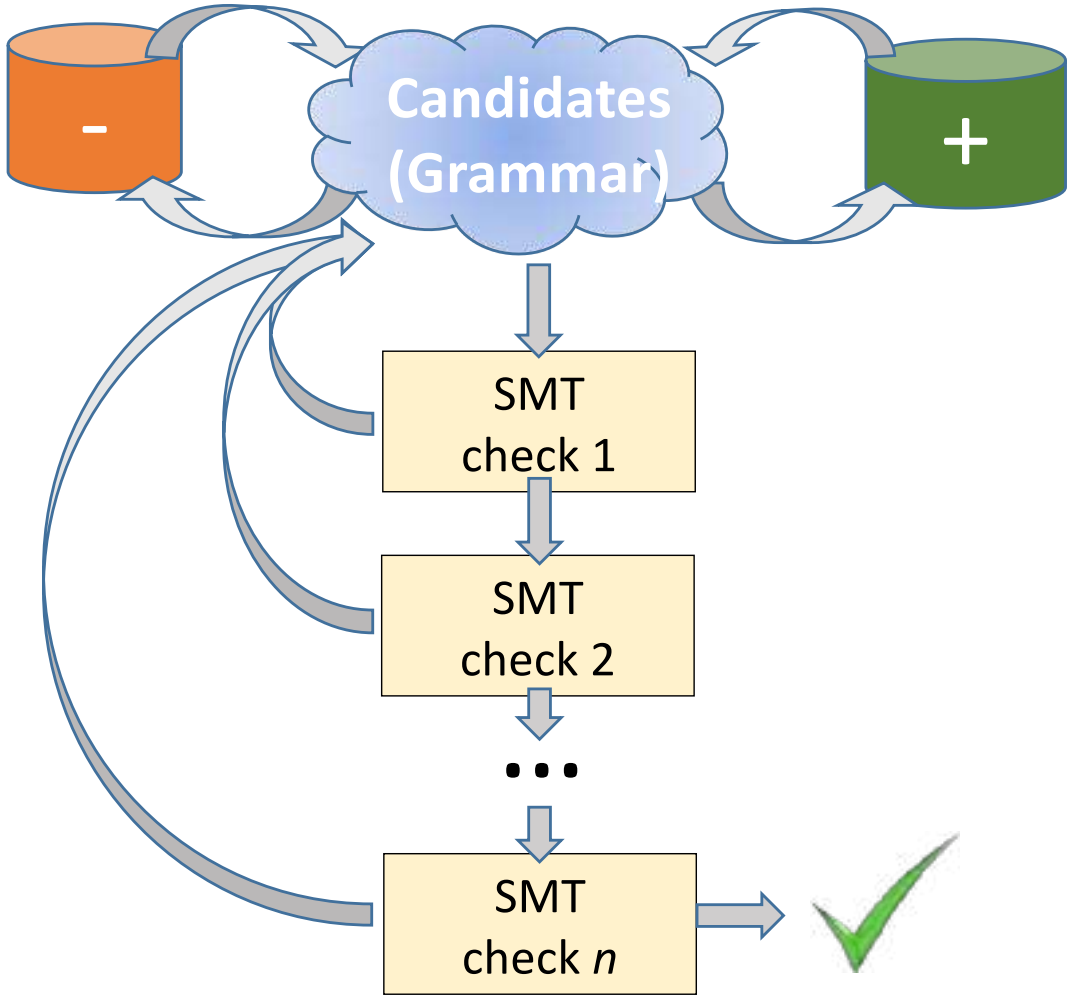
[Credit: Dana Fisman, SyGuS lectures]

- SyGuS has been used very successfully in many domains
  - Sketch, FlashFill, super-optimization, Scala^Z3, invariant generation, …

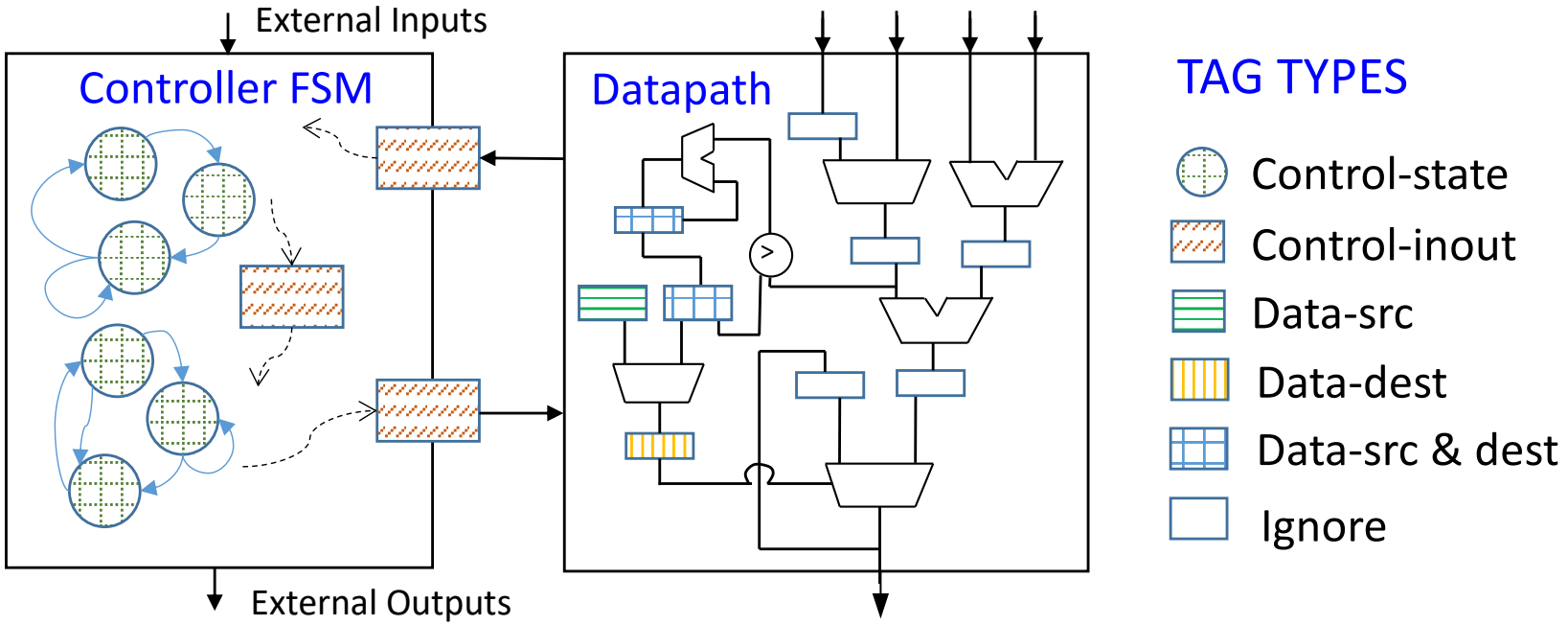
… but not for invariant generation on large hardware designs

Guess-and-check Invariants

- High-level view
  - Iterative flow with a candidate generator and an SMT-solver
- Candidate generator
  - Syntax-Guided Synthesis (SyGuS) grammar
  - Learning from positive / negative candidates
- SMT-based verifier
  - Off-the-shelf SMT solvers to verify whether a candidate is valid



Domain Insights via SyGuS Grammar

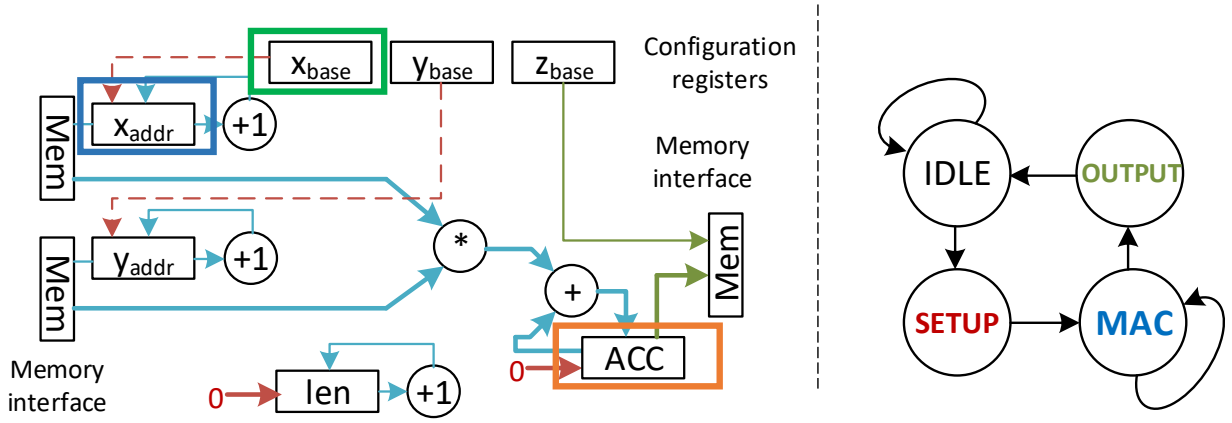


- RTL design structure
  - can be separated into Controller FSM and Datapath
  - tag all “state” variables in RTL (Verilog) with “type” (manually so far)

- Construct candidates from grammar on types
  - grammar defined on word-level variables (not bits)

An Example

Example of a vector dot-product machine: z = x.y



(a) Datapath

(b) Control-FSM

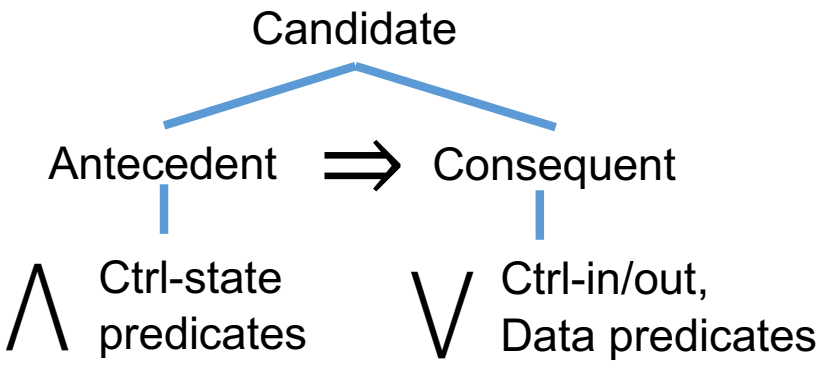
Tags

Tags	Ctrl-state	state
	Ctrl-inout	<none>
	Data-src	x_{base}, y_{base}, z_{base}, len
	Data-dst	x_{addr} y_{addr}, len
	Ignore	ACC

Example: Tags and Invariants

Pre-set Grammar in Grain

$$\langle \text{Cand} \rangle ::= \langle \text{CSpred} \rangle \implies \langle \text{Dpred} \rangle$$
$$\langle \text{CSpred} \rangle ::= \langle \text{CSvar} \rangle = \langle \text{Const}_C \rangle \mid \langle \text{CSvar} \rangle \neq \langle \text{Const}_C \rangle$$
$$\langle \text{Dpred} \rangle ::= \langle \text{DDvar} \rangle = \langle \text{DSvar} \rangle + \langle \text{DSvar} \rangle$$
$$\langle \text{Const}_C \rangle ::= 00 \mid 01 \mid 10 \mid 11$$



Tags: given by User

Tags	Ctrl-state	state
	Ctrl-inout	<none>
	Data-src	x_{base}, y_{base}, z_{base}, len
	Data-dst	x_{addr}, y_{addr}, len
	Ignore	ACC
Invariants		$state \neq IDLE \rightarrow x_{addr} = x_{base} + len$ $state \neq IDLE \rightarrow y_{addr} = y_{base} + len$

Invariants: generated by Grain

Grain: Tagging Statistics

- 2 synthetic examples: RC (Counters), Simple Pipeline (SP)
- 3 real-world designs: AES, Gaussian Blur are publicly-available accelerators, PicoRV32 is a Risc-V processor

Benchmarks	RC	SP	AES	PicoRV32	Gaussian Blur
# state-bits	8	72	963 *	1817	4840 **
# word-level state-vars	2	16	14	149	176
# ctrl-state		4	3	30	4
# ctrl-inout		2	2	34	8
# data-src	2		2		
# data-dst	2	2	3		11
# groups		2		3	

* for AES, this model abstracts the round-level computation (verified separately)
** for Gaussian Blur, this model abstracts internal block RAMs and MACs.

Grain: Experimental Results

- Grain is able to discover invariants for *all* designs
- Better performance than other tools on real-world designs

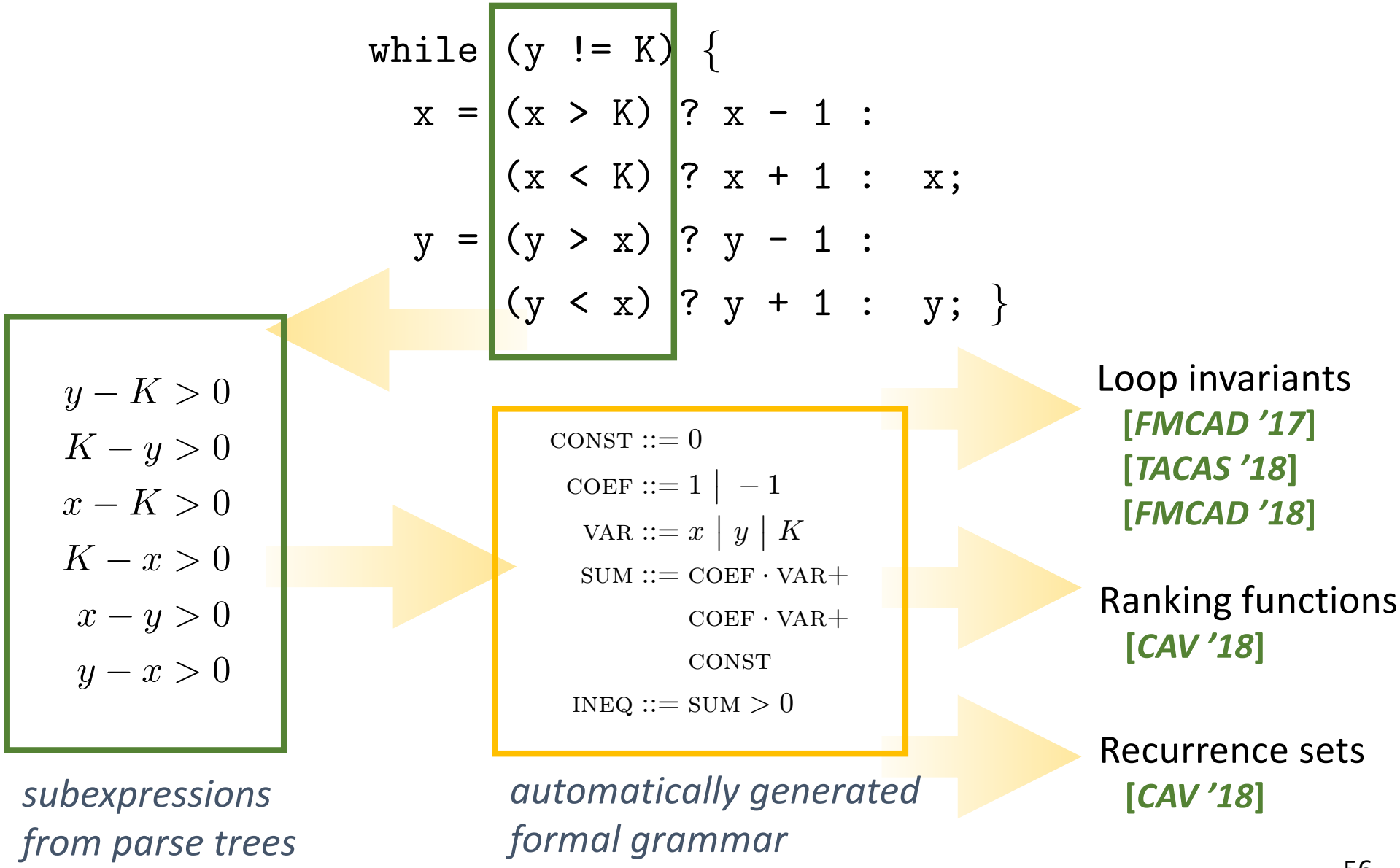
Solvers	ABC	Z3	Grain
RC	32.3s	16.2s	**2.0s**
SP	**29.7s**	40.4s	145.1s
AES	O.O.M	T.O.	**947.8s**
PicoRV32	O.O.M	7864.8s	**4429.4s**
Gaussian Blur	T.O.	2783.7s	**1046.0s**

O.O.M : out-of-memory (> 32GB)
T.O. : time-out (>10 hrs)

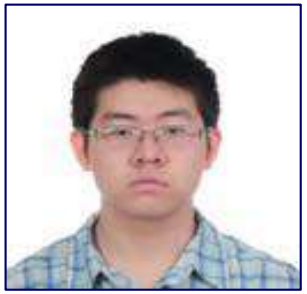
Grain: Summary

- ***Automated*** discovery of environment invariants
  - Reduces manual burden on designer
- Design insights from ***RTL structure*** in SyGuS grammar
  - Word-level invariants: easier to understand
  - Variable tagging with “types”
  - Grammar based on typed variables to achieve the desired “shape” of invariants
- Design structure leveraged in candidate generation
  - Candidate enumeration
  - Candidate filtering heuristics
- Future direction
  - Combine grammar-based guesses with deductive techniques

SyGuS for Program Verification



Application 3
Automating Induction over Algebraic Data Types



William Yang

Grigory Fedyukovich

Weikun Yang, Grigory Fedyukovich, Aarti Gupta: Lemma Synthesis for Automating Induction over Algebraic Data Types. CP 2019: 600-617

Proving Quantified Theorems by Induction

Motivating example

Definition of ADT List

$$\begin{aligned} \texttt{List} &:= nil \\ &:= cons(\texttt{Int}, \texttt{List}) \end{aligned} \tag{1}$$

Axioms for *concat*

$$\begin{aligned} &\forall l. concat(nil, l) = l \\ &\forall l_1, l_2, n. concat(cons(n, l_1), l_2) = cons(n, concat(l_1, l_2)) \end{aligned} \tag{2}$$

Axioms for *rev*

$$\begin{aligned} &rev(nil) = nil \\ &\forall n, l. rev(cons(n, l)) = concat(rev(l), cons(n, nil)) \end{aligned} \tag{3}$$

Goal

$$\boxed{\forall l. rev(rev(l)) = l}$$

Proof by Induction – base case

Goal $\forall l.rev(rev(l)) = l$

Base case

$$\begin{aligned} \text{LHS} &= rev(rev(nil)) \\ &= rev(nil) \\ &= nil \\ &= \text{RHS} \end{aligned}$$

Trivial rewrite using base case of Axiom (2)

$$\frac{Q(x) \equiv (P(x) = R(x))}{goal[P \mapsto R](x)} \text{ [rewrite]}$$

Proof by Induction – inductive case

Goal

$$\forall l.rev(rev(l)) = l$$

Inductive Hypothesis

$$rev(rev(l_1)) = l_1$$

Inductive case sub-goal

$$\boxed{\forall n.rev(rev(cons(n, l_1))) = cons(n, l_1)}$$

Proof Search (using Rewriting)

Root (ind. case sub-goal)

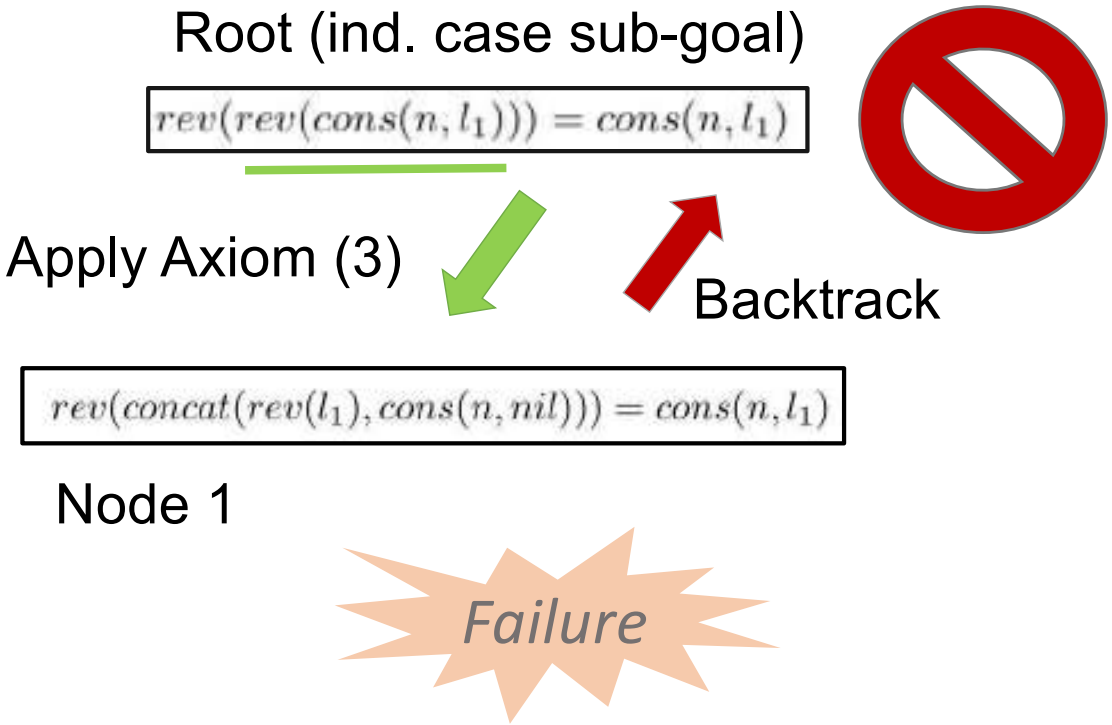
$$rev(rev(cons(n, l_1))) = cons(n, l_1)$$

Apply Axiom (3)

$$rev(concat(rev(l_1), cons(n, nil))) = cons(n, l_1)$$

Node 1

Proof Search (using Rewriting)



What if … a lemma appeared by magic?

Root (ind. case sub-goal)

$$rev(rev(cons(n, l_1))) = cons(n, l_1)$$

Apply Axiom (3)

$$rev(concat(rev(l_1), cons(n, nil))) = cons(n, l_1)$$

Node 1

Apply Magic Lemma

$$concat(rev(cons(n, nil)), l_1) = cons(n, l_1)$$

Node 2

Magic Lemma

$$\forall l_1, l_2. rev(concat(rev(l_1), l_2)) = concat(rev(l_2), l_1)$$

Complete proof sequence

Magic Lemma

$$\forall l_1, l_2. rev(concat(rev(l_1), l_2)) = concat(rev(l_2), l_1)$$

Root (ind. case sub-goal)

$$rev(rev(cons(n, l_1))) = cons(n, l_1)$$

Apply Axiom (3)

$$rev(concat(rev(l_1), cons(n, nil))) = cons(n, l_1)$$

Apply Magic Lemma

$$concat(rev(cons(n, nil)), l_1) = cons(n, l_1)$$

Apply Axiom (3) then base case of (2)

$$concat(cons(n, nil), l_1) = cons(n, l_1)$$

Apply Axiom (2) then base case of (2)

$$cons(n, l_1) = cons(n, l_1)$$

✔ Proved!

What is the magic?!

$$rev(concat(rev(l_1), cons(n, nil))) = cons(n, l_1)$$

Candidate lemma

$$\forall l_1, l_2 . rev(concat(rev(l_1), l_2)) = concat(rev(l_2), l_1)$$

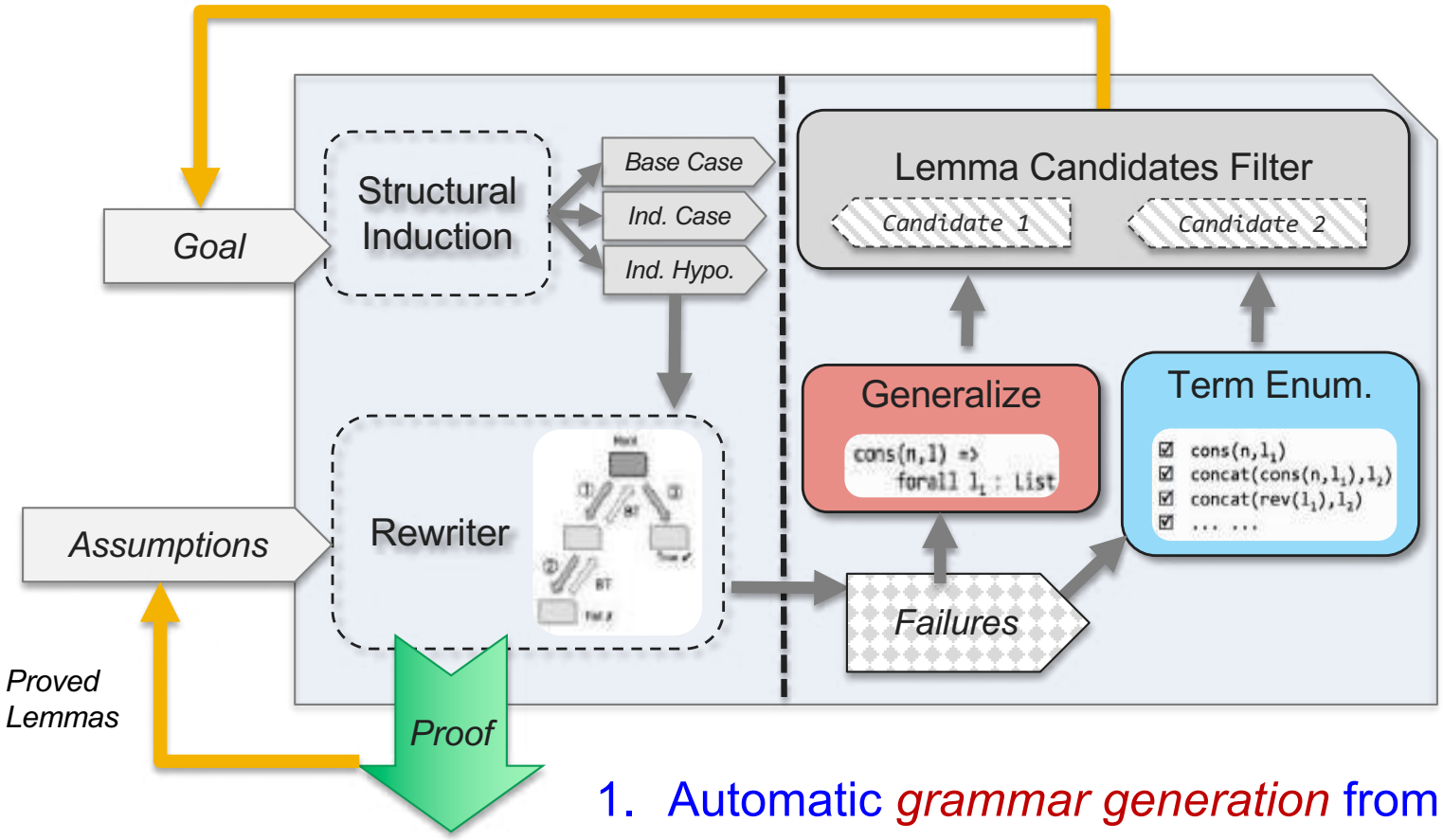
1. Generalize sub-terms

   e.g., cons(n, nil) to l_2

2. Enumerate candidate terms on RHS

   with all functions and variables

Overview of AdtInd



1. Automatic *grammar generation* from failures
2. *Lemma synthesis* using SyGuS
3. Apply smart filtering to prune the search space in term enumeration

Generalize from Failures

Replace base constructor

Replace inductive constructor

Replace function application

Function Def:

$$\forall l.\, len(nil) = 0$$
$$\forall l, n.\, len(cons(n, l)) = 1 + len(l)$$

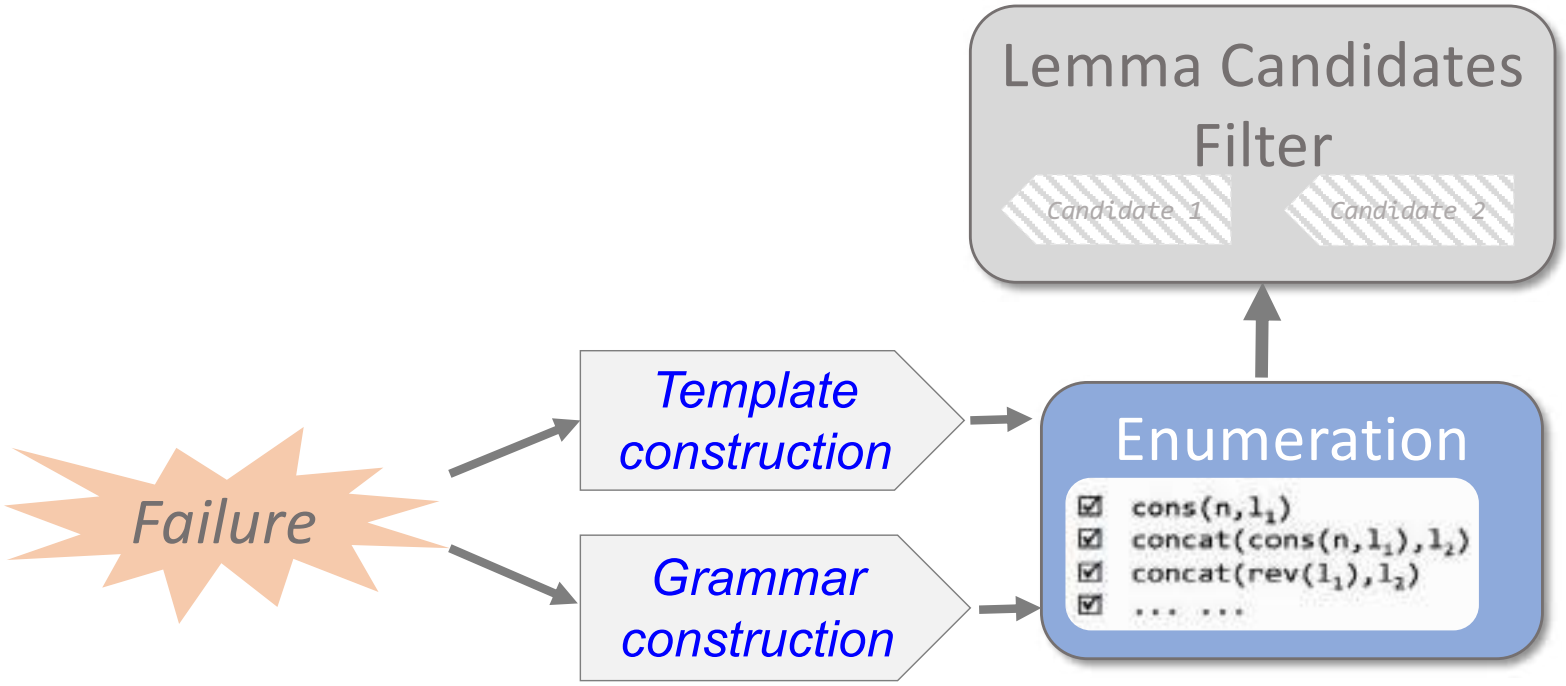
Goal: $\forall x, y.\, len(rev(concat(x, y))) = len(x) + len(y)$

Failure: $len(concat(rev(x), cons(n, nil))) = 1 + len(x)$

Replace function application and inductive constructor with new quantified variables

Required lemma: $len(concat(l_1, l_2)) = len(l_1) + len(l_2)$

Term Enumeration



Term Enumeration

φ : largest sub-term among failures (after generalization)

Templates

$$\varphi = \langle ??? \rangle + \langle ??? \rangle \quad \text{Int and Nat}$$

$$\varphi = \langle ??? \rangle \quad \text{Other data types}$$

$$\langle ??? \rangle = \langle ??? \rangle \quad \text{Fallback}$$

Grammar for terms

$$\langle \textbf{int-term} \rangle ::= 0 \mid 1 \mid \ldots$$

$$\langle \textbf{list-term} \rangle ::= nil \mid l_1 \mid l_2 \mid cons(\langle \textbf{int-term} \rangle, \langle \textbf{list-term} \rangle) \mid rev(\langle \textbf{list-term} \rangle) \mid concat(\langle \textbf{list-term} \rangle, \langle \textbf{list-term} \rangle)$$

$$rev(concat(rev(l_1), cons(n, nil))) = cons(n, l_1)$$

Generalize LHS to create template

$$rev(concat(rev(l_1), l_2)) = <?? >$$

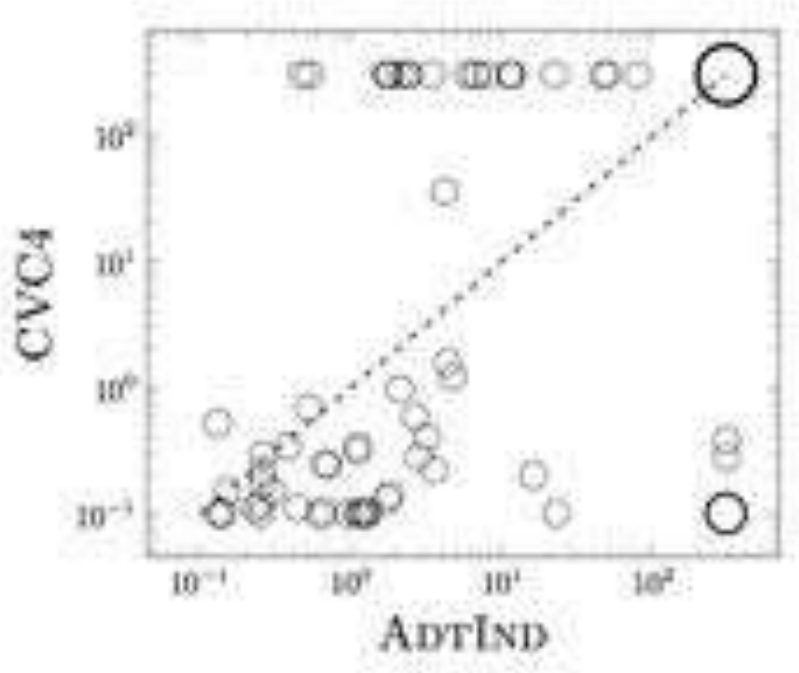
Enumerate terms to fill in RHS
Filter and refute

AdtInd Evaluation

Comparison with CVC4

on CLAM benchmarks

86 Nat, List theorems



AdtInd: proved 62 / 86
CVC4 : proved 47 / 86

Above diagonal line: AdtInd was faster

Top line: AdtInd solved in time whereas CVC4 timed-out (5min)

AdtInd Evaluation

9 ADT+ LIA theorems

Goal	AdtInd result	Goal	AdtInd result
list_rev	Proved, 10.6s	list_rev2_len	Proved, 0.95s
list_rev_concat	Proved, 2.52s	queue_push	Proved, 33.9s
list_rev2_concat	Proved, 1.95s	queue_len	Proved, 7.6s
list_rev2	Proved, 1m59s	tree_insert_all	Proved, 1.9s
list_rev_len	Proved, 2.30s		

Comparison with CVC4 and ACL2 (interactive)

CVC4: failed on list_rev, TO for rest

ACL2: proved 2 (list_rev_concat and list_rev_len)

Theme today: Summary

Approach 1: use a suitable encoding that matches well with the structure of the domain model

Approach 2: use domain insights to guide search for invariants in verification

- Exploit domain structure in models, theories, solver heuristics, verification techniques
- SyGuS grammars are a great way to systematically capture and leverage domain insights

Thanks to …

Ryan Beckett, Grigory Fedyukovich, Bo-Yuan Huang,
Ratul Mahajan, Sharad Malik, Pramod Subramanyan, Yakir Vizel,
David Walker, Yue Xing, Weikun Yang, Hongce Zhang

Thank you!