IC3 with Internal Signals

FMCAD'21

Rohit Dureja, IBM
Arie Gurfinkel, University of Waterloo
Alexander Ivrii, IBM
Yakir Vizel, The Technion

In this talk…

- IC3 is a *very powerful algorithm* for formal hardware verification
  - Especially for *proving* safety properties
- IC3 computes *an inductive invariant in CNF over state variables*
- But there are designs *without a concise invariant over state variables*
- *We extend traditional IC3* to learn invariants not only in terms of state variables but *also in terms of internal signals*
- The proposed method can learn *significantly more compact invariants* than IC3, while maintaining a highly-efficient CNF representation

Safety Verification

$S = \langle I, X, Init(X), Tr(I, X, X'), Bad(X) \rangle$

- I – primary inputs, X – state variables,
- $Init(X)$ – predicate defining the initial states, $Tr(I, X, X')$ – predicate defining the transition relation, $Bad(X)$ – predicate defining unsafe states,
- X' – state variables in the next-frame

The problem is UNSAFE if and only if there exists a *path* from an Init-state to a Bad-state, that is there exists a natural number N such that the following formula is satisfiable:

$$Init(X_0) \wedge Tr(I_0, X_0, X_1) \wedge \ldots \wedge Tr(I_{N-1}, X_{N-1}, X_N) \wedge Bad(X_N)$$

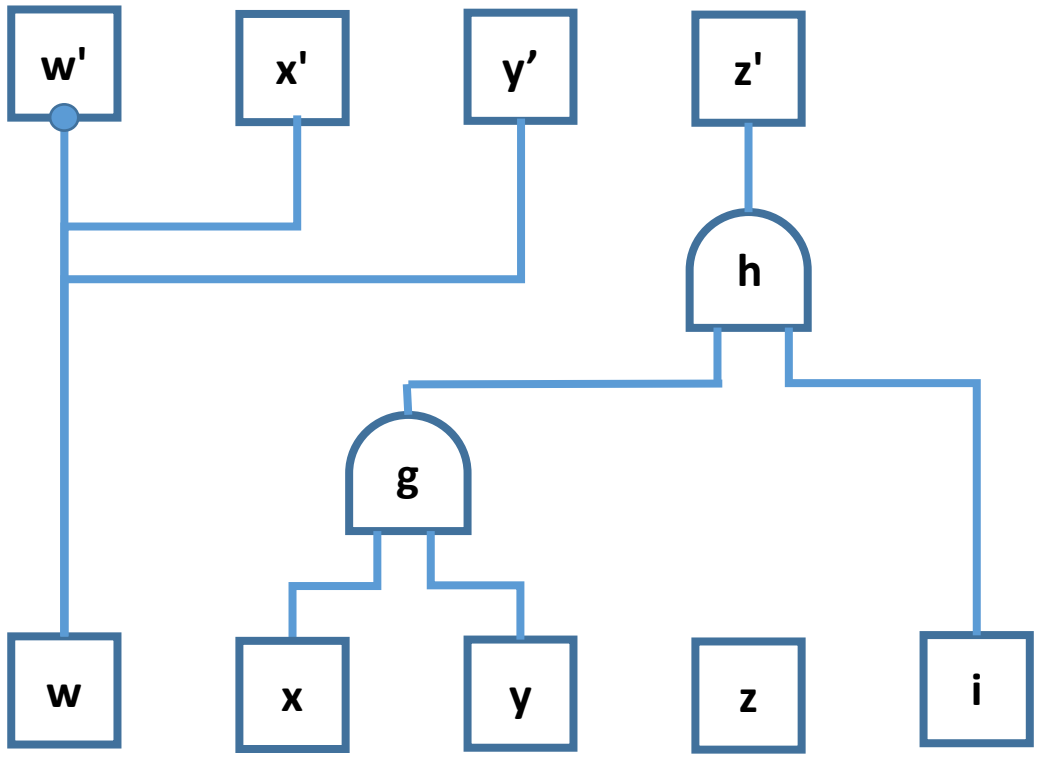
The problem is SAFE if and only if there exists a *safe inductive invariant* $Inv(X)$, that is

$$Init(X) \Rightarrow Inv(X)$$
$$(Inv(X) \wedge \exists I \,.\, Tr(I, X, X')) \Rightarrow Inv(X')$$
$$Inv(X) \Rightarrow \neg Bad(X)$$

Safety Verification

In hardware verification, Init, Tr, Bad are represented as netlists

Example:



Transition Relation as a netlist

(-w', -w), (w', w)

(-x', w), (x', -w)

(-y', w), (y', -w)

(-z', h), (z', -h)

(-h, g), (-h, i), (h, -g, -i)

(-g, x), (-g, y), (g, -x, -y)

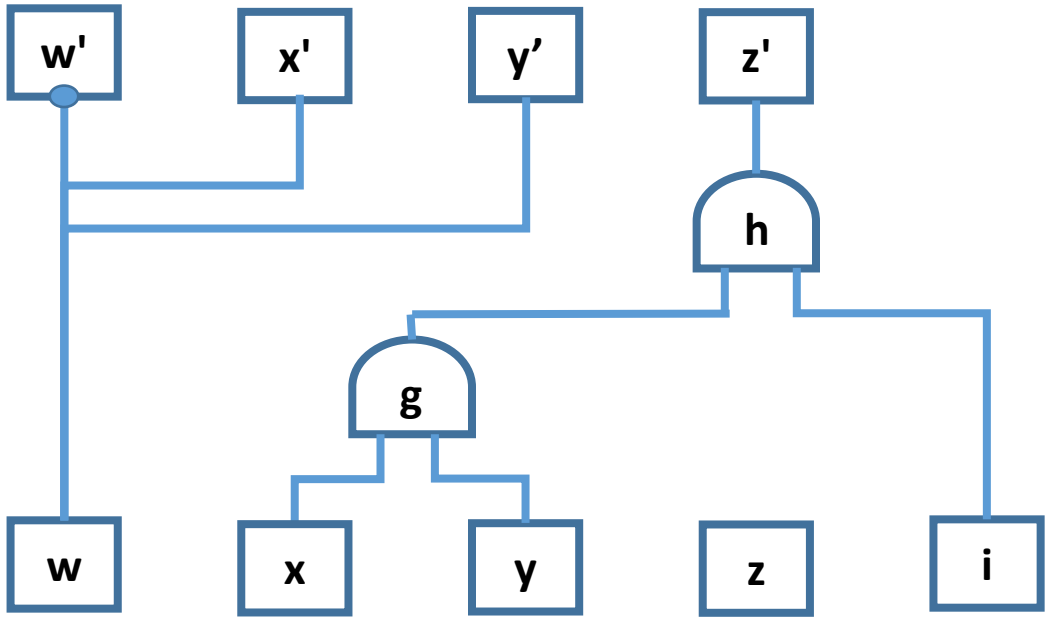
Transition Relation as CNF

Notation

Net: either a state variable (e.g., w, x, y, z), an input (e.g., i), or a logic gate (e.g., g, h) in the netlist

Latches: state variables and their negations

Internal nets (innards): internal logic gates and their negations



Example 1 (parity)

This is our favorite toy example

- State variables: $x_1, \ldots, x_n$
- Reachable states: $\{x_1, \ldots, x_n \mid x_1 \oplus \ldots \oplus x_n = 1\}$
- Unsafe states: $\{x_1, \ldots, x_n \mid x_1 \oplus \ldots \oplus x_n = 0\}$

The only safe inductive invariant *over latches* has 2^{n-1} clauses representing $x_1 \oplus \ldots \oplus x_n = 1$ in CNF:

- For example, there are 4 lemmas for n=3: $(x_1 \vee x_2 \vee x_3)$, $(\neg x_1 \vee \neg x_2 \vee x_3)$, $(\neg x_1 \vee x_2 \vee \neg x_3)$, $(x_1 \vee \neg x_2 \vee \neg x_3)$
- Every lemma blocks only a single state and cannot be generalized

Yet, for the innard $z = x_1 \oplus \ldots \oplus x_n$, there is a safe inductive invariant *over latches and innards* consisting of a *single* lemma:

- $(z=1)$

* A full example (in SMV and AIG formats), as well as other examples, are available at: https://github.com/agurfinkel/innard-benchmarks/

Example 2

This example is described *"Sequential Verification with Reverse PDR"* by Seufert and Scholl

- Two counters that count modulo-2^n, with state bits $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{n-1})$, respectively
- Input i
- Initial states: $\{ s \neq t \}$
- Transition relation: when $i=0$, both counters keep their values; when $i=1$ both counters increment by 1 modulo 2^n
- Bad states: $\{s = t\}$

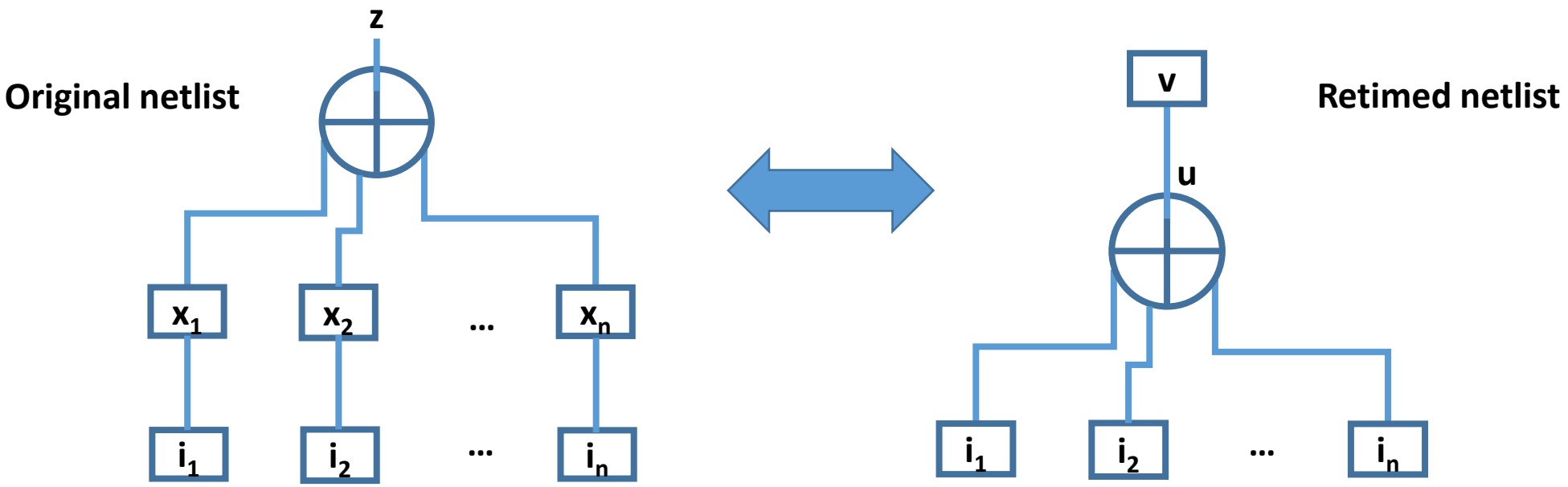
The above paper argues:

- Any safe inductive invariant *over latches* must contain at least 2^n lemmas
- There is a much smaller safe inductive invariant for the Reverse IC3
  - $2n$ lemmas that are required to represent $s = t$ in CNF

- For the innard $z = (s \neq t)$, there is an inductive invariant consisting of a *single* lemma *over latches and innards*
  - $(z=1)$

Example 3

This example illustrates a *sequential equivalence checking* problem between an original and a retimed design



- Unsafe states: { $z \neq v$}
- 2^n lemmas *over latches*
- 2 lemmas *over latches and innards*: $(\neg z \vee v)$, $(z \vee \neg v)$

Example 4

- Latches: $x_1, \ldots, x_n, y_1, \ldots, y_n$
- Innards: $z_1 = x_1 \wedge y_1, \ldots, z_n = x_n \wedge y_n$

Assume the lemma $C = (z_1 \vee \ldots \vee z_n)$ is inductive

Representing C in CNF requires 2^n lemmas.

For example, for n=3: $(z_1 \vee z_2 \vee z_3)$ is equivalent to 8 lemmas:

- $(x_1 \vee x_2 \vee x_3), (x_1 \vee x_2 \vee y_3), (x_1 \vee y_2 \vee x_3), (x_1 \vee y_2 \vee y_3),$
  $(y_1 \vee x_2 \vee x_3), (y_1 \vee x_2 \vee y_3), (y_1 \vee y_2 \vee x_3), (y_1 \vee y_2 \vee y_3)$

These lemmas are over different sets of variables!

The example is motivated by the benchmark rast-p16 from HWMCC'20

A very brief description of IC3: key data structures

IC3 maintains sets of clauses F_0, F_1, F_2, … called an *inductive trace*. Each F_k is called a *frame*. Each clause $C \in F_k$ is called a *lemma*. IC3 maintains the following invariant:

1. $F_0 = \text{Init}$
2. $\text{clauses}(F_{k+1}) \subseteq \text{clauses}(F_k)$; in particular, $F_{k+1} \Rightarrow F_k$
3. $F_k \wedge \text{Tr} \Rightarrow F_{k+1}'$

Intuitively, an inductive trace stores over-approximations of the states reachable in $1,2,3$, … steps from the initial states.

IC3 also maintains a queue of *proof obligations*. A single *proof obligation* is a pair <m, k>, where:

- m is a cube over (a subset of) latches
- $k > 0$ is the obligation's *level*

Intuitively, proof obligations are the states that need to be removed from the approximations in order to prove safety.

Both proof obligations and lemmas are clauses *over latches*

A very brief description of IC3: lemmas

Relative inductive: a lemma φ is *inductive relative* to a set of lemmas G iff

1. $Init \Rightarrow \varphi$,
2. $G \wedge Tr \wedge \varphi \Rightarrow \varphi'$

Learning new lemmas: At each point of execution, IC3 considers a proof obligation <m, k> and makes

- An initial query: SAT? $(Init \wedge \neg m)$
  - Checks whether a state in m in an initial state
- A predecessor query: SAT? $(\neg m \wedge F_{k-1} \wedge Tr \wedge m')$
  - Checks whether a state in m can be reached in one step from a state in F_{k-1}

When both queries are unsatisfiable, then $Init \Rightarrow \neg m$ and $F_{k-1} \wedge Tr \wedge \neg m \Rightarrow \neg m'$, i.e. $\neg m$ is inductive relative to F_k. In this case, IC3 can add the lemma $\neg m$ to all F_j for $j \leq k$

Inductive generalization: For performance, it is crucial to generalize $\neg m$ first: finding a shorter lemma $\varphi \subseteq \neg m$ such that $Init \Rightarrow \varphi$ and $F_{k-1} \wedge Tr \wedge \varphi \Rightarrow \varphi'$

- Performed by removing literals from $\neg m$ while the two conditions remain satisfied

These learned lemmas are used in future predecessor checks and pushing/convergence checks

A very brief description of IC3: pushing and convergence

Pushing: IC3 periodically pushes all lemmas:

- Given a lemma φ in $F_k \setminus F_{k+1}$, it checks if φ can be added to F_{k+1} as well

Convergence:

- If at any point there is k such that $F_k = F_{k+1}$ and $F_k \Rightarrow \neg Bad$, then we can take $Inv = F_k$ as a safe inductive invariant

Extending IC3 to reason about innards

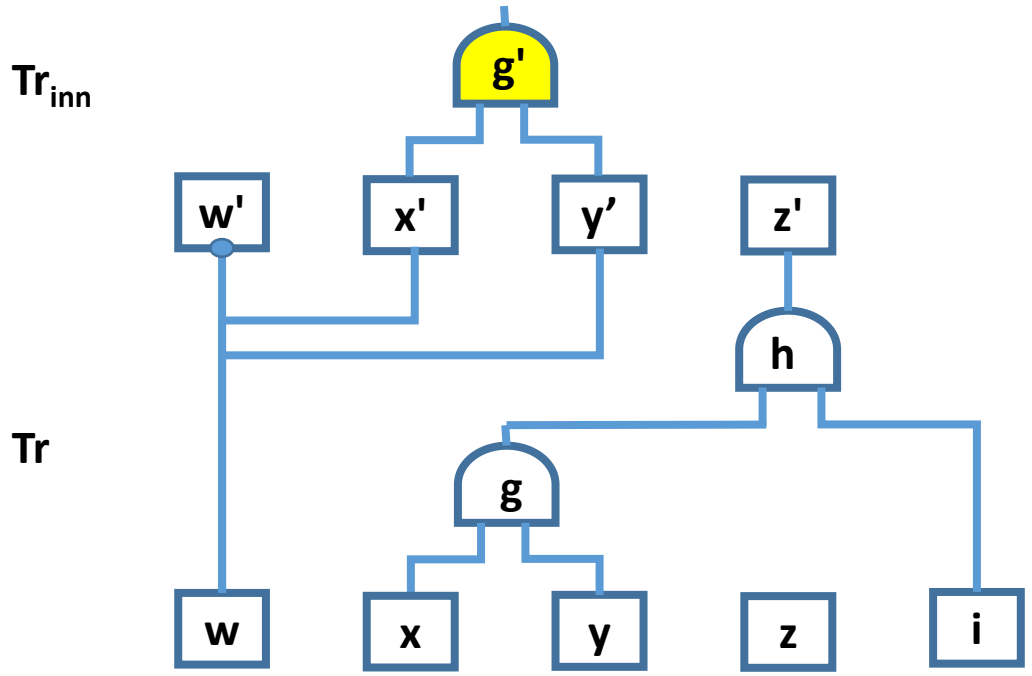
So far, we talked about the traditional IC3, and the concepts of a *safe inductive invariant*, *inductive trace*, *relatively inductive* were with respect to lemmas *over latches*.

But what needs to be changed to reason about innards (such as g)*?*
Fortunately, not much.

In the example, we need to reason about g in the initial state and the next state. For instance, we need to make sure that the SAT solver used for making inductive generalization queries includes the clauses defining g'.

However, once we let Tr_{inn} be the part of Tr that defines innards and define $INIT = Init \wedge Tr_{inn}$ and $TR = Tr \wedge Tr_{inn}'$, then replacing Init by INIT and Tr by TR in all definitions simply works!

- *There is something to prove here*, especially for innards that have inputs in combinational COI (such as h)



Proposed approach: learning lemmas over latches and innards

Proof obligations are *over latches* (the same as in traditional IC3)

Recall: given a proof obligation <m, k+1> *over latches* such that $\neg m$ is inductive relative to F_k:

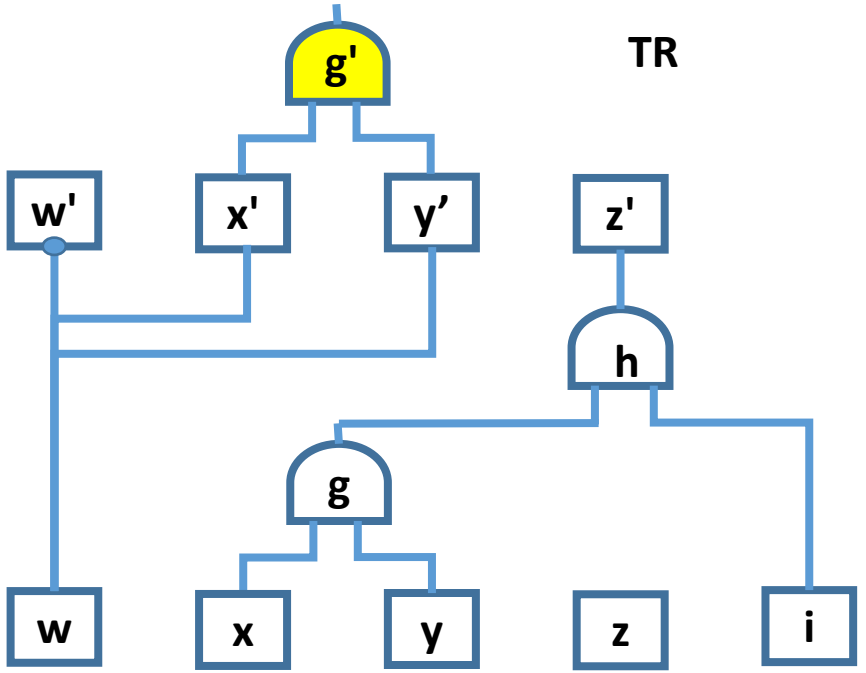
- Traditional IC3 computes a lemma φ *over latches* by inductively generalizing $\neg m$ with respect to F_k
- This lemma φ is then added to all frames F_j for $j <= k+1$

We construct *an additional lemma* ψ *over latches and innards* that is inductive relative to F_k

- This additional lemma ψ is also added to all frames F_j for $j <= k+1$
  (except if ψ is the same as φ)

In this way, *the proposed approach is a form of inductive generalization*

Our running example



Init: $(w=1) \wedge (x=1) \wedge (y=1) \wedge (z=1)$

Tr: $(w' = \neg w) \wedge (x' = w) \wedge (y' = w) \wedge (z' = h) \wedge (h = g \wedge i) \wedge (g = x \wedge y)$

Let the set of considered innards be $\{ g \}$

Let's suppose the original lemma is $\varphi = (w \vee x)$ and $F_k = T$

- φ is *over latches*
- φ is inductive relative to T

Recall: this means that

- $Init \Rightarrow \varphi$
- $\varphi \wedge Tr \Rightarrow \varphi'$
  - Indeed, $(w \vee x) \wedge TR \wedge \neg w' \wedge \neg x'$ is unsatisfiable

First step: extending lemma

φ and $\varphi 1$ *are equivalent modulo* Tr_{inn}

Given a lemma φ *over latches* that is inductive relative to F_k,
we extend φ to a lemma $\varphi 1 = (\varphi \vee \varphi 0)$ *over latches and innards* such that $Tr_{inn} \Rightarrow (\varphi \equiv \varphi 1)$

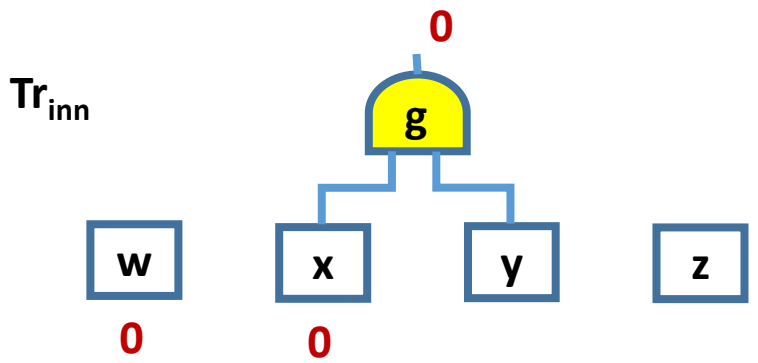
Idea:

- Find innards z such that $z \Rightarrow \varphi$ (modulo Tr_{inn}) and replace φ by $(\varphi \vee z)$
- Equivalently, find innards z such that $\neg\varphi \Rightarrow \neg z$
- Use constant propagation in the Tr_{inn} netlist

In our example:

- Start with $\varphi = (w \vee x)$
- Set $w = 0$, $x = 0$
- Constant propagation implies $g = 0$

Can replace φ by $\varphi 1 = (w \vee x \vee g)$



Remarks:

- It is easy to see that $\varphi 1$ remains inductive relative to F_k
- Extending lemmas with literals that imply it is closely related to *asymmetric literal addition* in SAT

Second step: generalizing the obtained lemma

Given a lemma $\varphi 1$ *over latches and innards* that is inductive relative to F_k,
Inductively generalize $\varphi 1$ by removing literals, *while prioritizing removal of latches / shallower nets*

Idea:

- Sort the literals of $\varphi 1$ according to their *logic level* (latches have the lowest level, deeper nets have higher level)
- Remove literals *in the given order* one-by-one as long as the lemma remains relatively inductive

In our example:

- Start with $\varphi 1 = (w \vee x \vee g)$ which is inductive relative to T; the literals are already sorted
- Cannot remove w as $(x \vee g)$ is not relatively inductive
- Can remove x as $(w \vee g)$ is relatively inductive; shrink to $(w \vee g)$
- Cannot remove g as (w) is not relatively inductive

Can replace $\varphi 1$ by $\varphi 2 = (w \vee g)$

Remark:

- This is an expensive procedure, but *see the paper for certain useful optimizations*

Our running example

To recap:

- We started with the lemma $\varphi = (w \vee x)$ *over latches* constructed by traditional IC3
- We extended it $\varphi 1 = (w \vee x \vee g)$ *over latches and innards*
- We obtained $\varphi 2 = (w \vee g)$ by inductive generalization, $\varphi 2$ is *over latches and innards*

Remarks:

- As $g = x \wedge y$, the lemma $\varphi 2 = (w \vee g)$ is equivalent to $(w \vee x) \wedge (w \vee y)$
  - The new lemma over latches and innards is equivalent to *two* different lemmas over latches
  - These two lemmas are over different sets of variables

- As φ is equivalent to $\varphi 1$ modulo Tr_{inn}, and as $\varphi 2$ is obtained by removing literals from $\varphi 1$, then $Tr_{inn} \Rightarrow (\varphi 2 \Rightarrow \varphi)$
  - $\varphi 2$ is stronger than φ (i.e., $\varphi 2$ blocks the same states *and maybe more* than φ)

Experimental Results

- Implemented in IBM's formal verification tool

- IC3 is the default variant of IC3 used by the tool
- IC3-INN is the variant with additional learning of lemmas over innards (with innards restricted to internal nets with no inputs in combinational COI)

- Benchmark sets are explained on the following slides; all the instances either are or expected to be *unsatisfiable*

TABLE I
SUMMARY OF EXPERIMENTAL RESULTS

benchmarks	#instances	time-limit per instance	IC3 solved (unique)	IC3 time	IC3-INN solved (unique)	IC3-INN time
IBM-AOD-SEC	3 605	300	2 562 (0)	424 885	3 605 (1 043)	**2 465**
6s119-SEC	364	600	364 (0)	2 906	364 (0)	**1 207**
6s22-SEC	310	600	262 (22)	32 701	278 (38)	**24 774**
AES-SEC	16	3 600	13 (0)	11 186	15 (2)	**5 601**
HWMCC11	278	3 600	277 (6)	**40 186**	272 (1)	55 557
HWMCC17	76	3 600	76 (0)	**7 963**	76 (0)	11 221
HWMCC20	192	3 600	190 (5)	**35 907**	187 (2)	41 448

IBM-AOD-SEC benchmarks

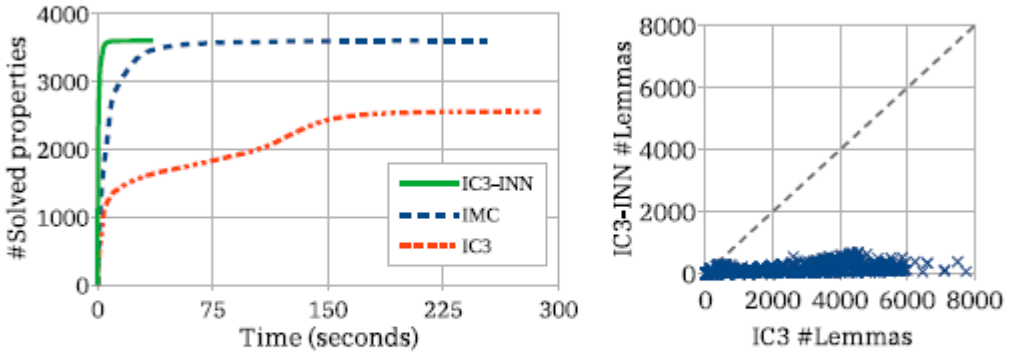
benchmarks	#instances	time-limit per instance	IC3 solved (unique)	IC3 time	IC3-INN solved (unique)	IC3-INN time
IBM-AOD-SEC	3 605	300	2 562 (0)	424 885	3 605 (1 043)	**2 465**

Benchmarks checking *sequential equivalence* between two designs in the Aspect Oriented Design flow at IBM

- Includes retiming and clock-gating

The SEC problem is *very challenging*, and traditionally solved using a combination of techniques:

- Using speculative reduction to reduce the problem into multiple simpler (but still hard) sub-problems
- Dedicated engine configuration consisting of combinational rewriting, k-induction, localization, and eventually a proof-based technique, such as Interpolation (IMC) or IC3



(a) Solved vs. Runtime (b) Invariant size

Fig. 3. Performance of IC3 and IC3-INN on AOD SEC benchmarks.

- Interpolation: generally, works well, but is not stable
- IC3 performs very poorly
- IC3-INN is amazing

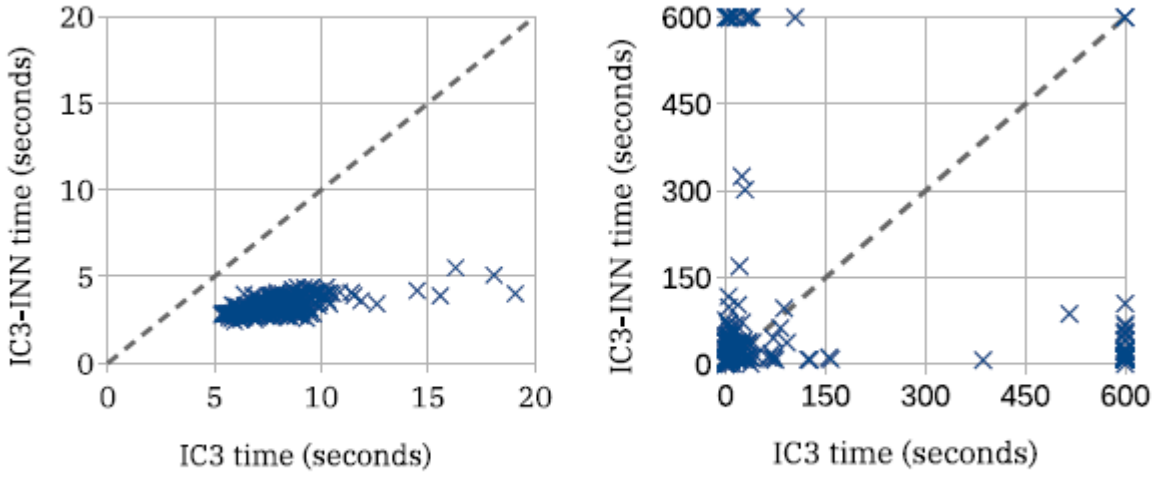
* unfortunately, this benchmark set cannot be released

6s119-SEC, 6s22-SEC benchmarks

benchmarks	#instances	time-limit per instance	IC3 solved (unique)	IC3 time	IC3-INN solved (unique)	IC3-INN time
6s119-SEC	364	600	364 (0)	2 906	364 (0)	**1 207**
6s22-SEC	310	600	262 (22)	32 701	278 (38)	**24 774**

Sequential equivalence checking benchmarks manually created from publicly available benchmarks

- Checking equivalence between the original design and the retimed designed
- Applying the sequential equivalence checking flow described previously
- Available at https://github.com/agurfinkel/innard-benchmarks/



(a) 6s119-SEC (b) 6s22-SEC

6s119-SEC:

- 364 rather easy problems
- IC3-INN is about 2.4x faster than IC3
- Interpolation: cannot solve 64 of these (within 600 s)

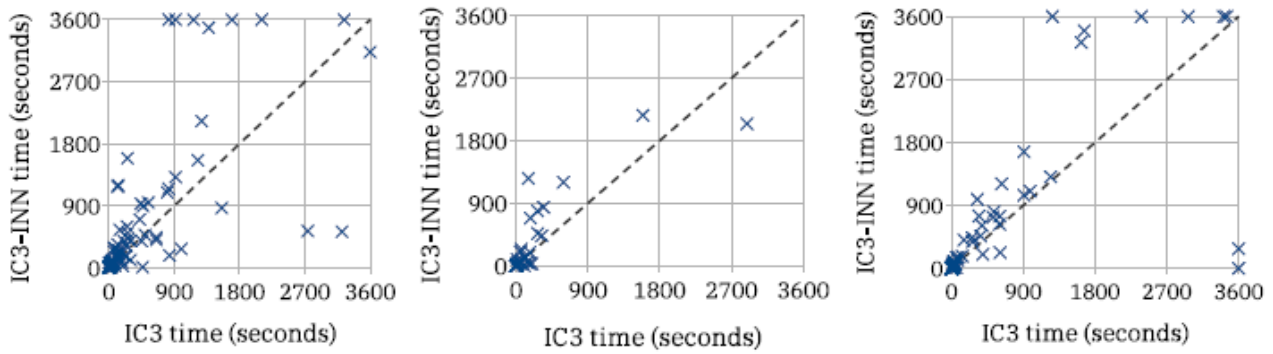
6s22-SEC:

- 310 problems
- IC3-INN solves 278 and IC3 solves 262 (within 600 s)
- Interpolation: significantly worse

HWMCC benchmarks

benchmarks	#instances	time-limit per instance	IC3 solved (unique)	IC3 time	IC3-INN solved (unique)	IC3-INN time
HWMCC11	278	3 600	277 (6)	**40 186**	272 (1)	55 557
HWMCC17	76	3 600	76 (0)	**7 963**	76 (0)	11 221
HWMCC20	192	3 600	190 (5)	**35 907**	187 (2)	41 448

Publicly available *Hardware Model Checking Competition Benchmarks*



(a) HWMCC'11 (b) HWMCC'17 (c) HWMCC'20

- In general, IC3-INN performs worse than IC3
- A few benchmarks where IC3-INN is significantly better than IC3
  - E.g., *rast-p16*: IC3 times out, IC3-INN solves within 2 s; exposes the pattern from Example 4

Related and Future Work

- In the presented approach, proof obligations are *over latches*, while lemmas are *over latches and innards*
- In the work *"Using cubes of non-state variables with property directed reachability"*, Backes and Riedel consider proof obligations *over certain internal signals*
  - Innards on the boundary between input-free and non input-free parts of the netlist
  - Allows to *generalize proof obligations*
- **Q**: The two approaches are quite different, but maybe one can somehow combine them?

- In the presented approach, the design is fixed, and *we use internal signals that are already available*
- An interesting direction to extend the approach to learn lemmas *over signals that are not present in the original netlist*
  - Closely related to *"IC3 modulo theories via implicit predicate abstraction"*, by Cimatti, Griggio, Mover and Tonetta
  - The framework allows such an extension
- **Q:** How to decide which additional logic/additional innards to include in the netlist

- What about learning lemmas *over arbitrary formulas*, not restricted to clauses in CNF?
  - Common in SMT-based extensions of IC3, such as Sally or Spacer
  - However, these techniques seem difficult to port efficiently to hardware model checking

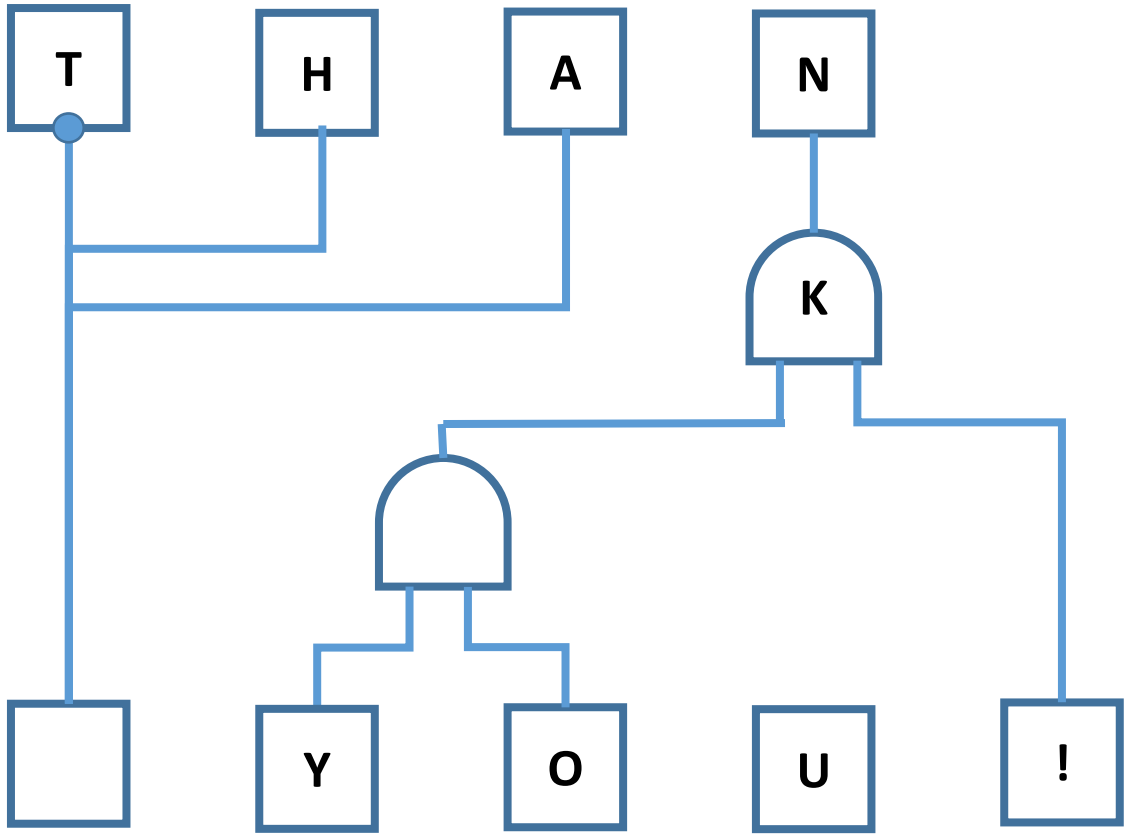
Conclusions

- The presented technique is an *extension* of regular IC3 that simply *learns an additional lemma during inductive generalization*

- *Easy to integrate in an existing IC3 implementation*
  - The main technical point is to replace Init by INIT and Tr by TR

- Currently, the technique seems more beneficial for (unsatisfiable) *sequential equivalence checking* benchmarks
  - We probably need more of such benchmarks in HWMCC competitions

- Opens many interesting questions and directions for further work


T
H
A
N
K
Y
O
U
!