DESIGN, AUTOMATION & TEST IN EUROPE
19 – 23 March, 2018 · ICC · Dresden · Germany
The European Event for Electronic System Design & Test

Efficient Verification of Multi-Property Designs (the benefit of wrong assumptions)

E. Goldberg, M. Güdemann, D. Kroening
Diffblue Ltd,
Oxford, UK

R. Mukherjee
Cadence Design Systems,
USA

Outline

- Motivation and problem definition
- “Just-assume” verification
- Experimental results
- Conclusions

Motivation

- Main bulk of research: single property verification
- A design can have thousands of properties
- A hard property $\Rightarrow$ conjunction of easier properties

Need for efficient methods of multiple-property verification

Problem Definition

- Given sequential circuit and *safety properties* $P_1,...,P_k$
  - check if every P_i is true
  - if some P_i fails $\Rightarrow$ design *is incorrect*

- *How many* failed properties does one need to find?
  - Straightforward approach:
    - find *every failed property* P_i
    - a flaw: same bug can break *many properties*
  - We take a more practical approach:
    - find a (small) subset of failed properties *identifying bugs*

Joint and Separate Verification

- Joint verification: check *aggregate property* $P := P_1 \wedge ... \wedge P_k$
  - design is correct iff P holds
- Separate verification: prove each P_i separately
  - P_i is weaker than $P \Rightarrow$ it should be *easier to prove*
  - different properties can have *quite different proofs*
  - inductive invariant for P_i *can be re-used* when proving P_m
- "Just-assume" verification: an instance of separate verification
  - verify P_i assuming that every P_m, $m \neq i$ holds
  - no justification of assumptions is necessary, hence the name

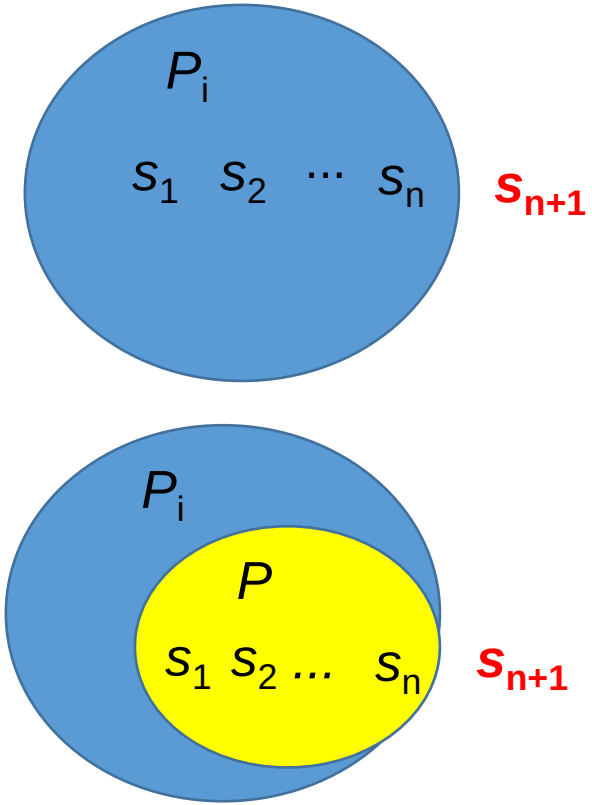
Background

- Using *design structure* to group similar properties
  - G.Cabodi, S.Nocco (DATE 2011)
  - P. Camurati, C. Loiacono, P. Pasini, D. Patti, S. Quer (DIFTS 2014)
  - G. Cabodi, P.E. Camurati, C. Loiacono, M. Palena. P. Pasini, D. Patti, S. Quer (Int J Software Tool Tech Tran, 2017)
- On-line information on multi-property verification by ABC
  - ABC implements joint verification
- HWMCC results, multi-track (up to 2013)

Outline

- Motivation and problem definition
- “Just-assume” verification
- Experimental results
- Conclusions

Proving Properties Globally and Locally

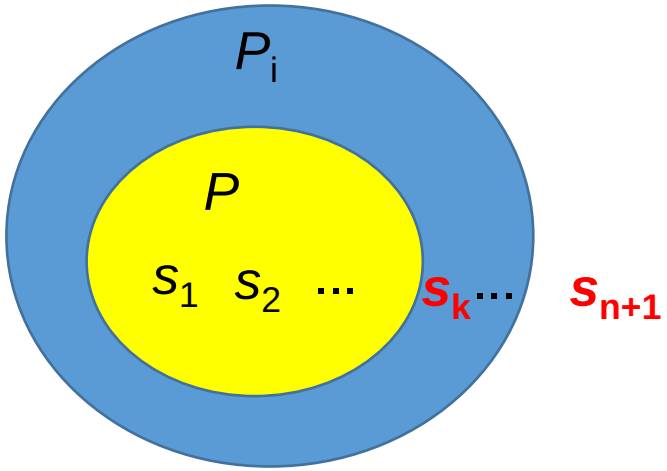


Proving P_i globally:
no CEX s_1,.., s_n,s_{n+1} where s_1,..,s_n are $\boldsymbol{P_i}$**-states** and s_{n+1} is ~P_i state

Proving P_i locally: (w.r.t $P := P_1 \wedge ... \wedge P_k$)
no CEX s_1,.., s_n,s_{n+1} where s_1,..,s_n are $\boldsymbol{P}$**-states** and s_{n+1} is ~P_i state

Proving P_i locally means assuming that *every* P_m, $m \neq i$ *holds*

Relation Between Global and Local Proofs



If P_i holds globally it does locally too

The opposite it not true

If P_i holds locally it either

- holds globally OR
- every CEX breaking P_i first breaks P_m

s_k is P_i-state and ~P-state

s_k breaks some property P_m

Advantage of Verifying Properties Locally

- Proving P_i locally is easier than P
  - *proving P* **:** can one reach **~*P*-state** by transitions from ***P*-states** ?
  - *proving P_i locally* **:** can one reach **~P_i-state** by transitions from ***P*-states** ?
  - the ~P_i-states is *a subset* of the ~P-states
- If P_i holds locally, it is most likely *not a bug-identifying property*
  - even if P_i fails globally, some property P_m fails before P_i
- If P_i fails locally, it is a *bug-identifying property*
  - there is a CEX where P_i is the first to fail
- If P fails $\Rightarrow$ at least one P_i fails locally (and hence globally)

Example

```
module counter (enable, clk, request);
  parameter reset_val = 1 << 7;
   input enable, clk, request;
  reg [7:0] val ;
  wire reset ;
  initial val = 0;
  assign reset = (( val == reset_val ) && request);
  always @( posedge clk ) begin
    if ( enable ) begin
      if (reset ) val = 0;
      else val = val +1;
    end
  end
endmodule
```

P_1: assert property (request == 1);
P_2: assert property (val <= reset_val);

Both P_1 and P_2 *fail globally*

Consider proving P_1 and P_2 *locally* with respect to P: = $P_1 \wedge P_2$

P_1 fails locally (i.e. assuming P_2 is true)

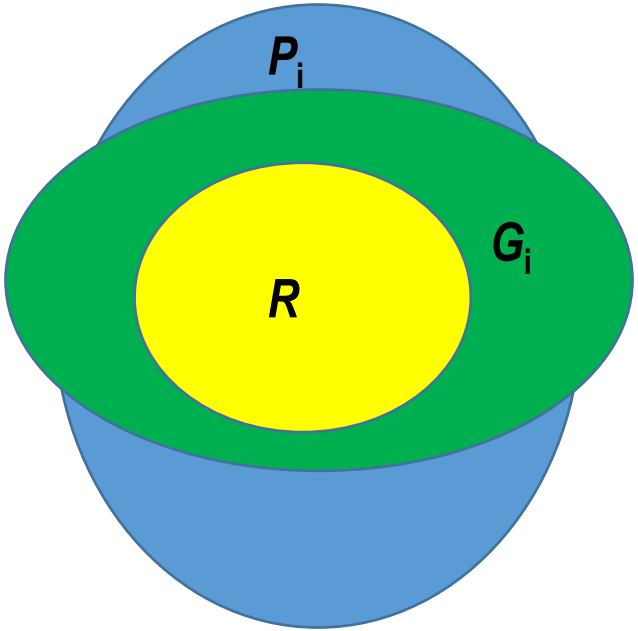
P_2 holds locally (i.e. assuming P_1 is true)

“Just-Assume” (Ja) Verification

- Check every property P_i locally
  - i.e. we assume that every P_m , $m \neq i$ holds
- If every P_i holds locally $\Rightarrow$ aggregate property P holds
  otherwise
- Properties failing locally identify bugs

- *No justification* of assumptions is required
- When proving P_i locally
  - assumption “P_m holds” is useful *even if it is wrong*:
  - we simply drop traces where P_m fails before P_i

Re-using Inductive Invariants



Let R be the set of *reachable states*

Proving P_i by induction:
Find strengthening G_i such that
$P_i \wedge G_i$ is an inductive invariant

Both P_i and G_i *over-approximate* R

Let $G_1,..,G_i$ be strengthenings for $P_1,..,P_i$
Proving $P_{i+1} \Rightarrow$ proving $G_1 \wedge .. \wedge G_i \wedge P_{i+1}$

Outline

- Motivation and problem definition
- “Just-assume” verification
- Experimental results
- Conclusions

Implementation of Ja-Verification

- In experiments, we used IC3-db, a Diffblue version of IC3
- To prove P_i locally, IC3-db treats P_m, $m \neq i$ *as constraints*
- Ja-verification was implemented as a Perl script
  - IC3-db is called in a loop to prove properties locally one by one
- Order in which properties are verified *matters*
  - the reason is re-using of inductive invariants
  - *a rule of thumb*: prove easy properties first
  - re-use inductive invariants when proving harder properties
- We verified $P_1,..,P_k$ in the order they were listed

Implementation of Joint Verification

- We also used IC3-db to implement joint verification
  - as a Perl script iteratively calling IC3-db
- Implementation is meant for solving all properties globally
- The script first calls IC3-db to check $P := P_1 \wedge ... \wedge P_k$
  - If P holds, all properties P_i are true

    otherwise
  - false properties are removed, remaining properties are conjoined
- We cross-checked results of IC3-db by ABC (UC, Berkeley)
  - Joint verification is a natural mode of operation for ABC

Comparison of Joint and Ja-verification

- Joint verification is less robust than separate verification
- Complexity of proving $P := P_1 \wedge ... \wedge P_k$ *blows up*
  - if a few properties P_i are *too hard to solve*
  - properties P_i depend on *different local behaviors*
- This problem can be solved by *clustering* similar properties
  - we want to make a *semantic comparison*
- We used HWMCC-13 benchmarks
  - *correct designs*: 8 designs solved by joint verification without clustering (under 1000 properties each)
  - *faulty designs*: 8 designs where at least one property was *proved false*

Designs with Failed Properties

Name	#lat-ches	#pro-per-ties	Joint verification				Ja-verification by IC3-db		
			ABC		IC3-db				
			false (tr)	time	false (tr)	time	time limit	false (tr)	total time
6s104	84,925	124	1 (0)	10 h	1 (0)	memout	0.3 h	1 (123)*	**2.5 h**
6s260	2,179	35	1 (0)	10 h	1 (0)	10 h	0.5 h	1 (34)*	**1,686 s**
6s258	1,790	80	25 (0)	10 h	30(0)	10 h	0.3 h	1 (72)	**2.4 h**
6s175	7,415	3	2 (0)	10 h	2 (0)	10 h	0.3 h	2 (1)*	**554 s**
6s207	3,012	33	6 (0)	10 h	10 (0)	10 h	0.3 h	2 (31)*	**22 s**
6s254	762	14	13 (1)*	25 s	13 (1)*	225 s	0.3 h	1 (13)*	**2 s**
6s335	1,658	61	26 (35)*	2 h	26 (35)*	260 s	0.3 h	20 (41)*	**56 s**
6s380	5,606	897	399 (0)	10 h	395 (0)	10h	0.3 h	3 (894)*	**550 s**

Correct Designs

Name	#lat-ches	#pro-per-ties	Joint verification		Ja-verification by IC3-db		
			ABC time	IC3-db time	time limit	#un-solved	total time
6s124	6,748	630	> 10 h	2.9 h	0.8 h	0	**1.9 h**
6s135	2,307	340	123 s	**335 s**	0.8 h	0	746 s
6s139	16,230	120	4.7 h	**1.7 h**	2.8 h	2	6.5 h
6s256	3,141	5	> 10h	**602 s**	2.8 h	1	2.9 h
bob12m09	285	85	1,692 s	930 s	0.8 h	0	**784 s**
6s407	11,379	371	1.3 h	3.4 h	0.8 h	0	**2,077 s**
6s273	15,544	42	1.8 s	325 s	0.8 h	0	**290 s**
6s275	3,196	673	334 s	**1,154 s**	0.8 h	0	1,611 s

Conclusions

- We introduce "Just-Assume" (Ja) verification
  - it is a special case of separate verification
- We give a *semantic version* of Ja-verification
  - a structure-aware method can be built *on top of it*
- In Ja-verification, assumptions *do not need justification*
- CEXs are built only for failed properties *identifying bugs*
  - this can give big performance gains (finding CEXs can be very hard)
- Joint and Ja-verification are *competitive* on correct designs