arm

Formal Verification at Arm

Polytech Sophia

May 2019

ARM Ltd is a subsidiary of SoftBank Robotics

Laurent Arditi, Senior Principal Engineer

© 2019 Arm Limited

Agenda

- About Arm
- Design verification
- Formal verification
- Algorithms (BDD, SAT, model checking)
- Applications
- Formal methods for software

© 2019 Arm Limited

From inception to now

1990

Joint venture between Acorn Computers and Apple.

Designed into first mobile phones and then smartphones.

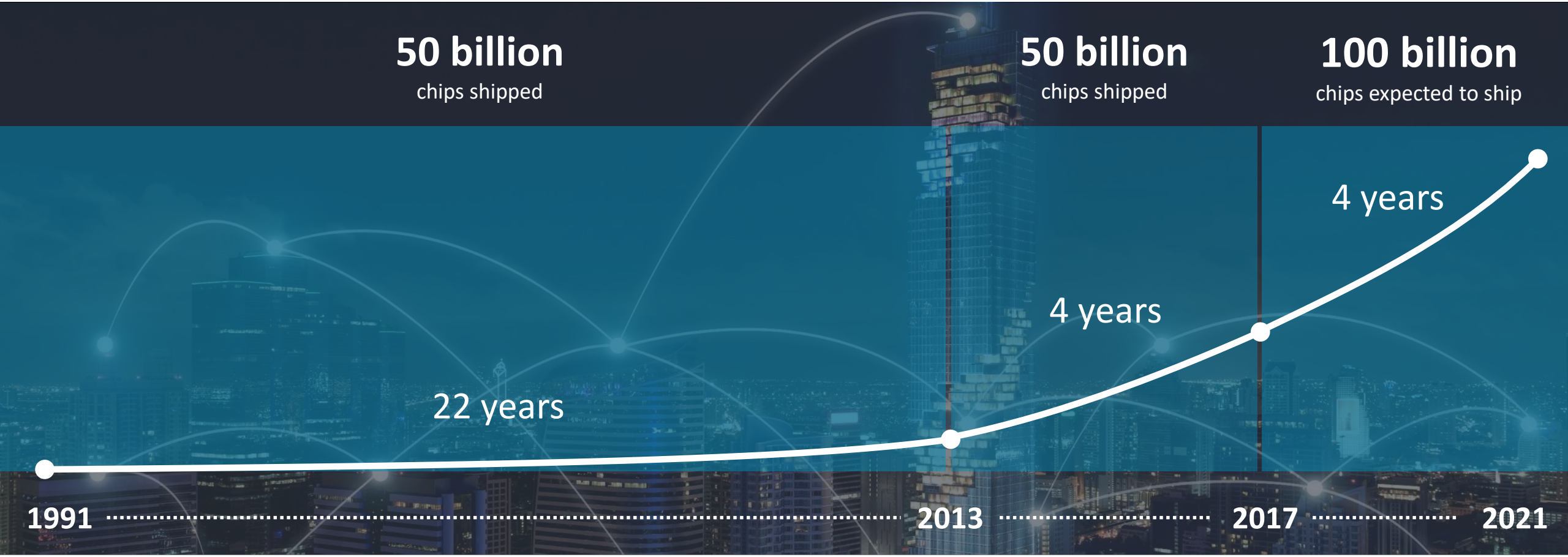
1993 onwards

Today

Now all electronic devices can use intelligent Arm technology.

© 2019 Arm Limited

Acceleration of Arm technology deployment



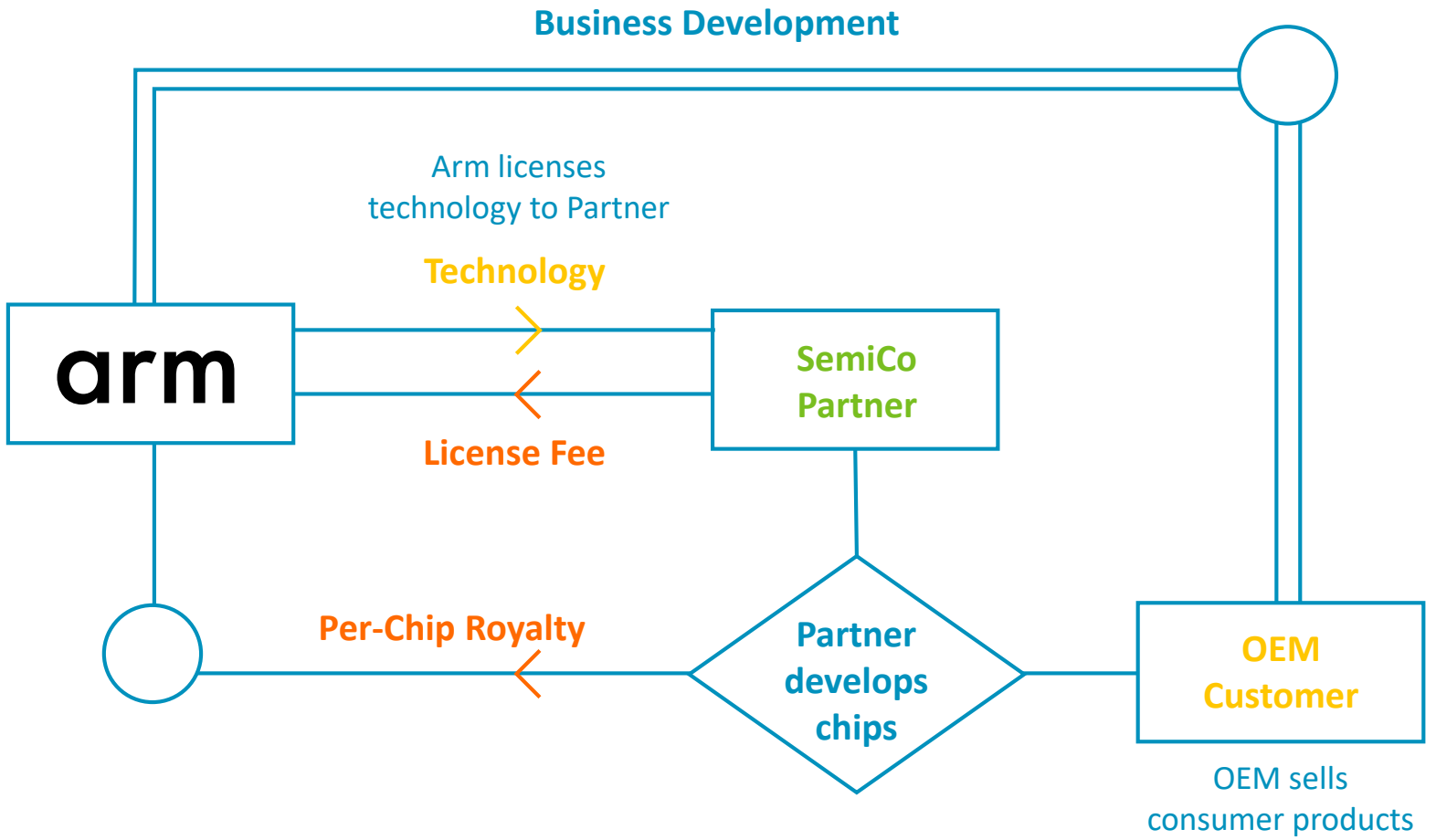
Devices with Arm processors

© 2019 Arm Limited

A continuous partnership model

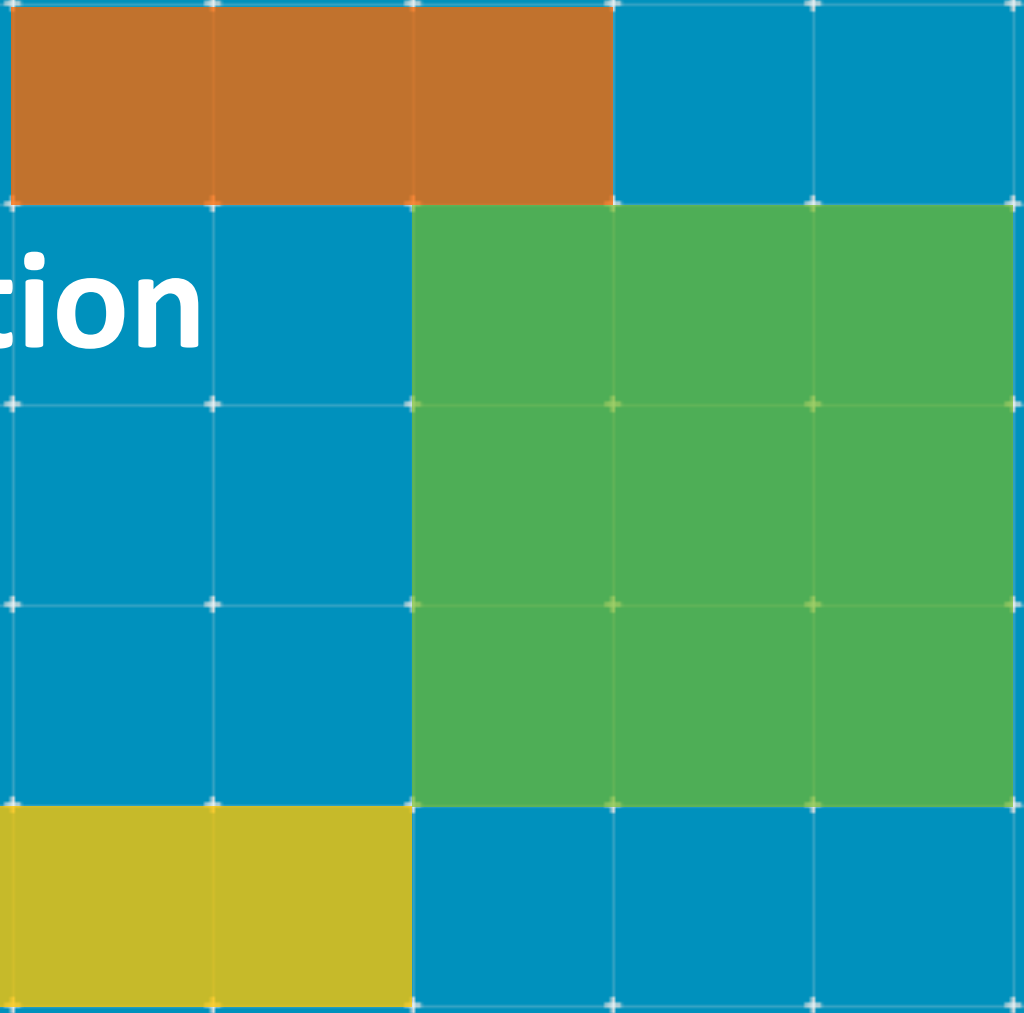
Arm develops technology that is licensed to semiconductor companies.

Arm receives an upfront license fee and a royalty on every chip that contains its technology.



© 2019 Arm Limited
arm

Design and verification at Arm



© 2019 Arm Limited

arm

Hardware design in one slide

How to build a CPU?

- Each CPU is an implementation of a defined (i.e. specified) “architecture”. E.g. Arm v8.
- The architecture specifies:
  - The “programmer view”: mostly registers
  - The encodings and semantics and each assembly instruction
  - Some required behaviours for caches, memory management, etc.
- Designers refer to the architecture to build the CPU “micro-architecture”, respecting performance/power/area requirements:
  - Pipeline stages,
  - Physical registers,
  - Arithmetic blocks,
  - Internal buffers, FIFOs,…
- Designers describe the micro-architecture using an Hardware Description Language such as VHDL or Verilog
  - Main available structures are registers, if-then-else, sequential or concurrent blocks, expressions on bit-vectors

© 2019 Arm Limited

How to verify a design?

Even for small designs the full possible state space can exceed the number of atoms in the universe!

- Theoretical Max State Space of a design is $2^{(\#\text{storage elements})}$
  - For a small design with only 1K registers, the theoretical max state space is 2^{1000}.
- Cycles to *exhaustively* verify 32-bit adder: 2^{64} (18 billion billion)
- Number of stars in universe: 2^{70} (10^{21})
- Number of atoms in the universe: 2^{260} (10^{78})

Of course, “exhaustive completeness” is not reality

- Nor a sensible approach to the problem
- And not all state space is reachable
- The problem must be tackled using a systematic, structured and measurable approach.

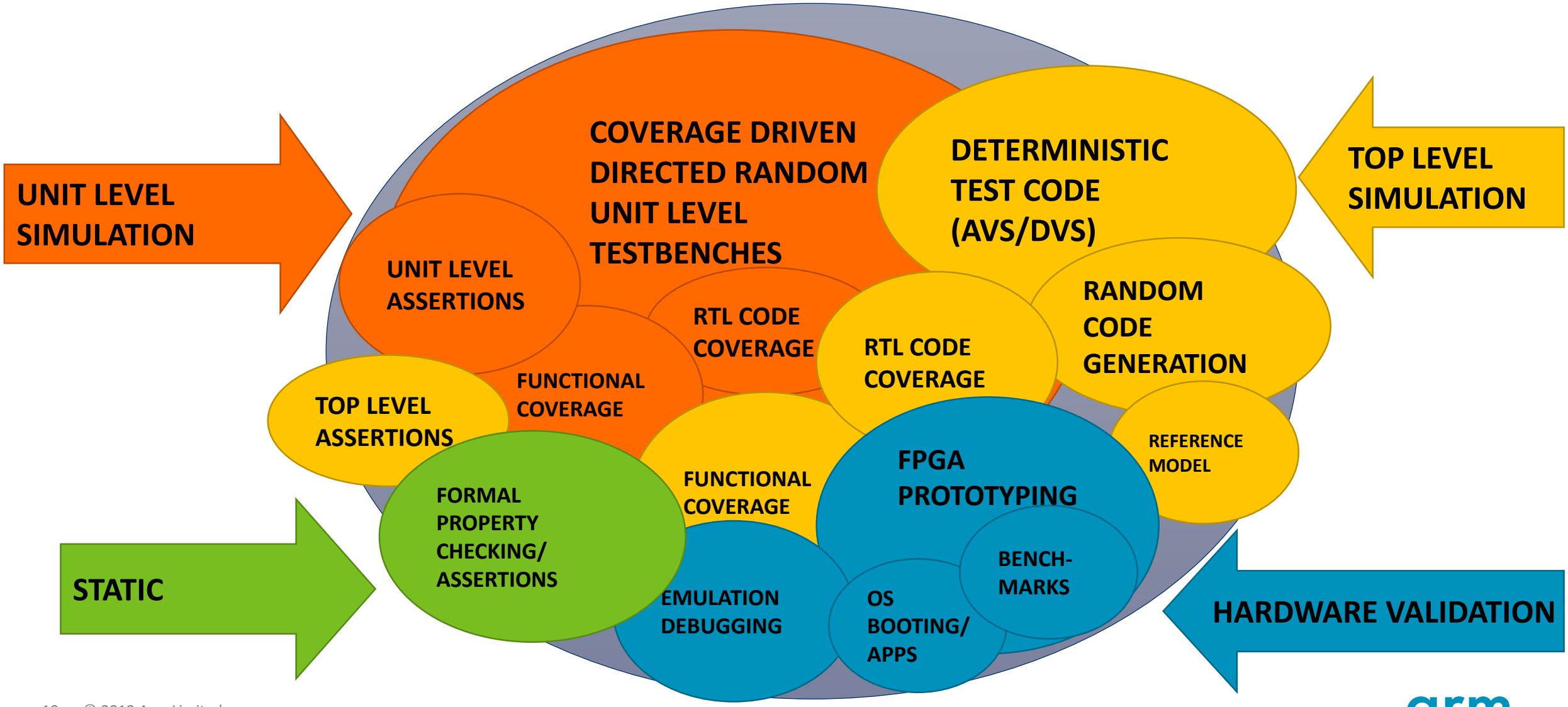
© 2019 Arm Limited

Multiple and complementary “Best Practices” are required

- Formal analysis: few cycles of exploration
  - But can prove properties
- Simulation speeds are approx 100Hz-2KHz
- Emulator speeds are approx 500KHz
- FPGA speeds are approx 4MHz
- Silicon speed is from 400MHz to few GHz
  - i.e. Silicon is able to do 1,000,000 times more work per second than simulation.
    - 12 days simulation for just 1 second of real time!
  - …and ~100 times more work per second than FPGA.

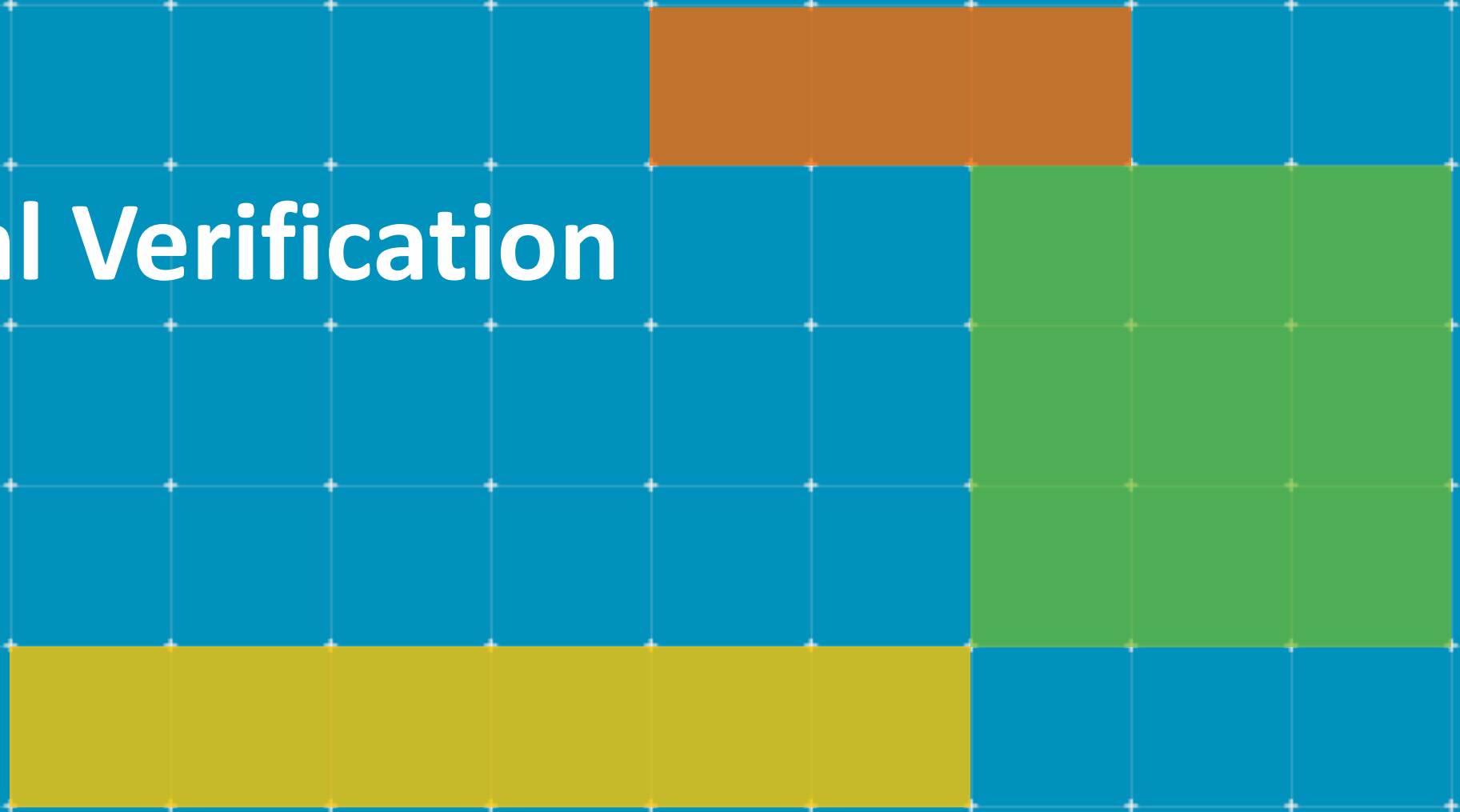
© 2019 Arm Limited

Multiple and complementary “Best Practices” are required



© 2019 Arm Limited
arm

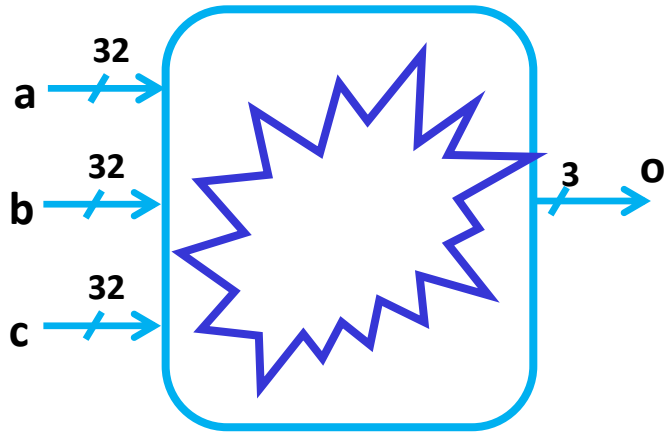
Formal Verification



© 2019 Arm Limited

arm

Which of these assertions are true?




```
module m (input wire [31:0] a,
          input wire [31:0] b,
          input wire [31:0] c,
          output wire [2:0] o);

assign o = {(c > a) & (~c > b),
            (a > 32'h12345678) & (c + b < 32'hacdc) & (b < 32'h87654321) & a[12],
            (b[26:5] > c[27:6]) & ((b[17:2] ^ c[27:12]) << 7) > (a[18:6] >> 5)};
```

```
p000: assert property(o != 3'b000);
p001: assert property(o != 3'b001);
p010: assert property(o != 3'b010);
p011: assert property(o != 3'b011);
p100: assert property(o != 3'b100);
p101: assert property(o != 3'b101);
p110: assert property(o != 3'b110);
p111: assert property(o != 3'b111);
```


© 2019 Arm Limited
arm

Use simulation-based verification

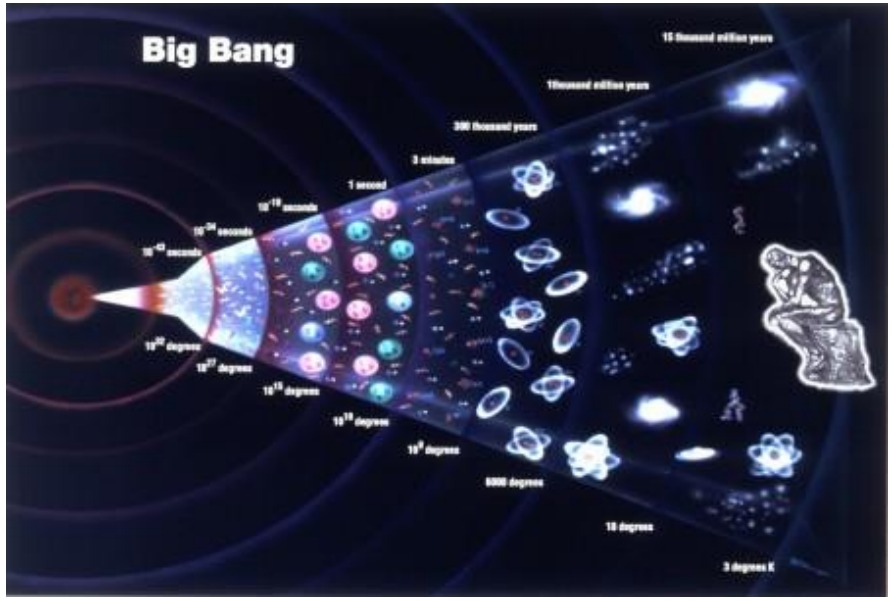
```
module m (input wire [31:0] a,
          input wire [31:0] b,
          input wire [31:0] c,
          output wire [2:0] o);

assign o = {(c > a) & (~c > b),
            (a > 32'h12345678) & (c + b < 32'hacdc) & (b < 32'h87654321) & a[12],
            (b[26:5] > c[27:6]) & ((b[17:2] ^ c[27:12]) << 7) > (a[18:6] >> 5)};
```

3x 32-bit wide inputs: 2^96 = 7.9x10^28 possible values.

If “simulating” one value per cycle on a 3Ghz CPU: 300 Trillions years of simulation.

For reference, the universe is only 14 Billions years old.



© 2019 Arm Limited
arm

Use formal verification

```
module m (input wire [31:0] a,
          input wire [31:0] b,
          input wire [31:0] c,
          output wire [2:0] o);

assign o = {(c > a) & (~c > b),
            (a > 32'h12345678) & (c + b < 32'hacdc) & (b < 32'h87654321) & a[12],
            (b[26:5] > c[27:6]) & ((b[17:2] ^ c[27:12]) << 7) > (a[18:6] >> 5)};
```

No testbench required, only a few tool commands

The runtime is too short to be measured

	Type	Name	Engine	Bound	Time
✗	Assert	m.p000	Hp	1	0.1
✗	Assert	m.p001	N	1	0.0
✗	Assert	m.p010	N	1	0.0
✗	Assert	m.p011	N	1	0.0
✗	Assert	m.p100	N	1	0.0
✗	Assert	m.p101	B	1	0.0
✓	Assert	m.p110	Hp (1)	Infinite	0.0
✓	Assert	m.p111	Hp (1)	Infinite	0.0

© 2019 Arm Limited
arm

What is Formal Verification?

"Formal" – applying **automated mathematical techniques** to reason about a design

Static testing (simulation is **dynamic**)

- No stimulus

Reason about **properties** of a design

- Property is logical statement about design
- Either true or false

Can give **exhaustive proofs**

- Property is always true for all possible cases

© 2019 Arm Limited

Model Checking

Industry in general, including Arm, uses various methods of Formal Verification

Main methodology is **Model Checking**

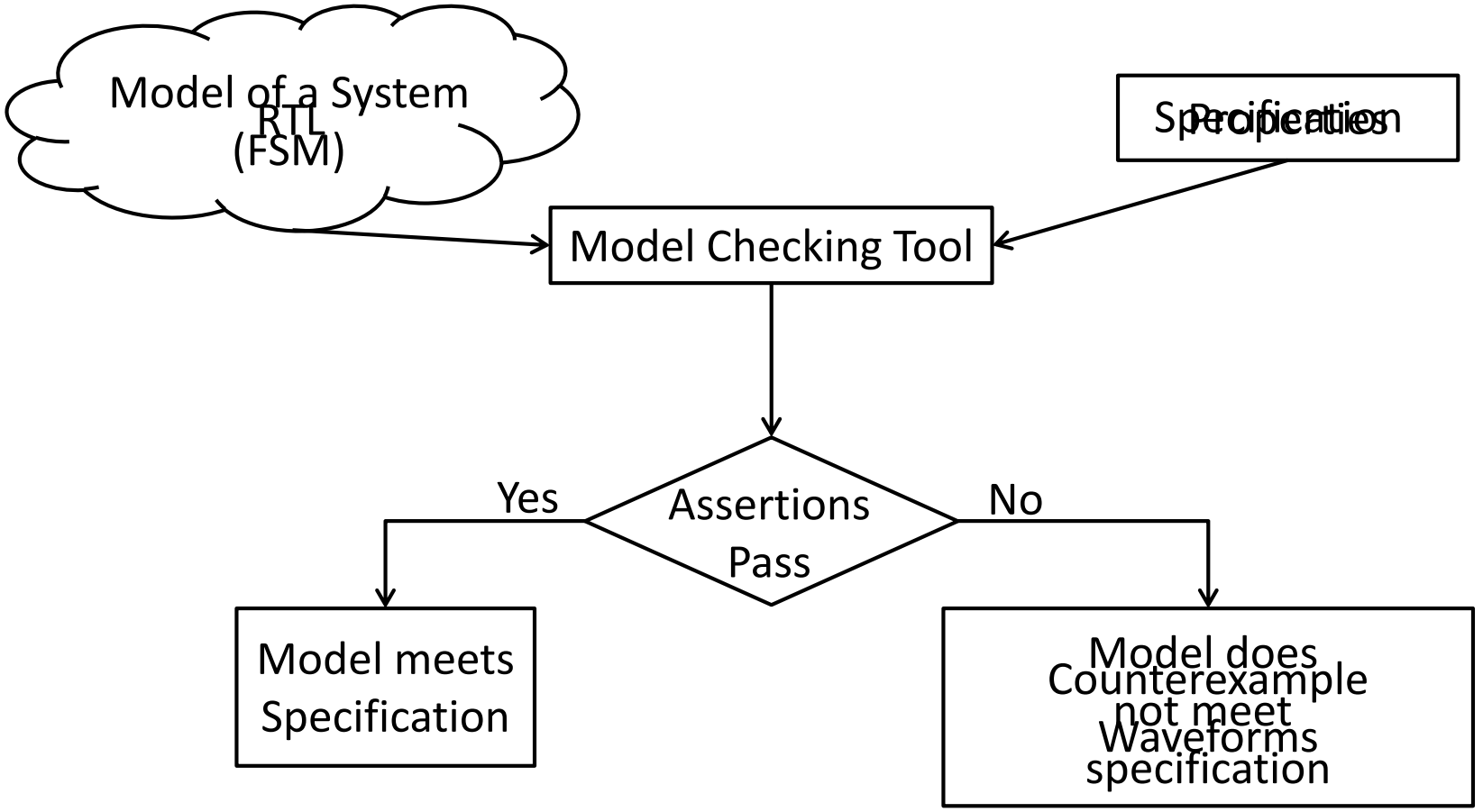
- Well supported by the EDA community

Wikipedia defines Model Checking as follows:

- “Given a model of a system, exhaustively and automatically check whether this **model** meets a given **specification**”

© 2019 Arm Limited

Model Checking



© 2019 Arm Limited
arm

In practice…

Engineer writes **properties** about design

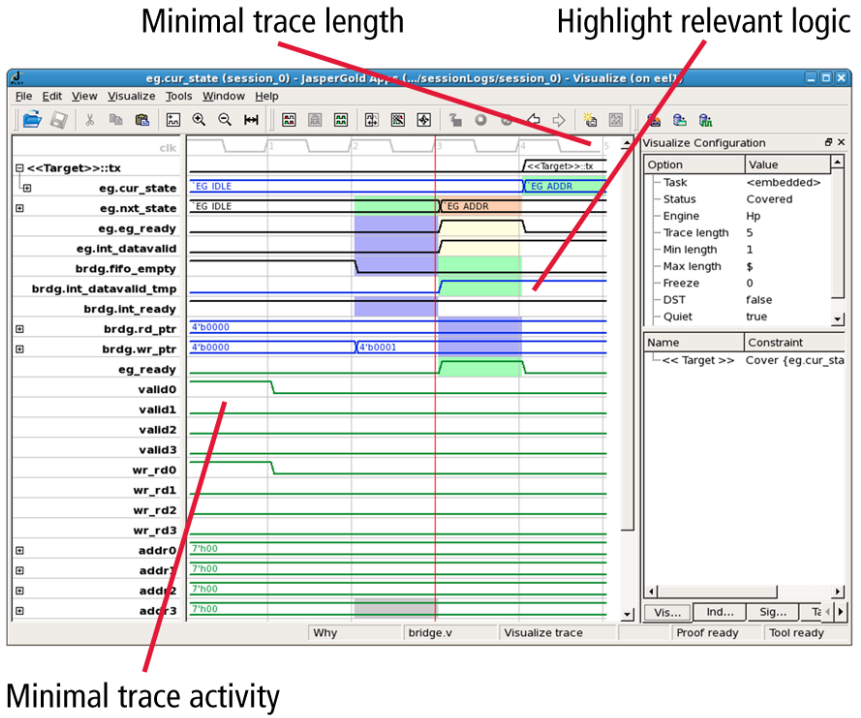
- "signal A is always high when signal B is high" `B |-> A`

Formal tool proves/disproves **asserted** property

- Either gives **proof** – "yes, signal A is always high when signal B is high"
- or **counterexample** – "no, here is a case where B is high but A is low"

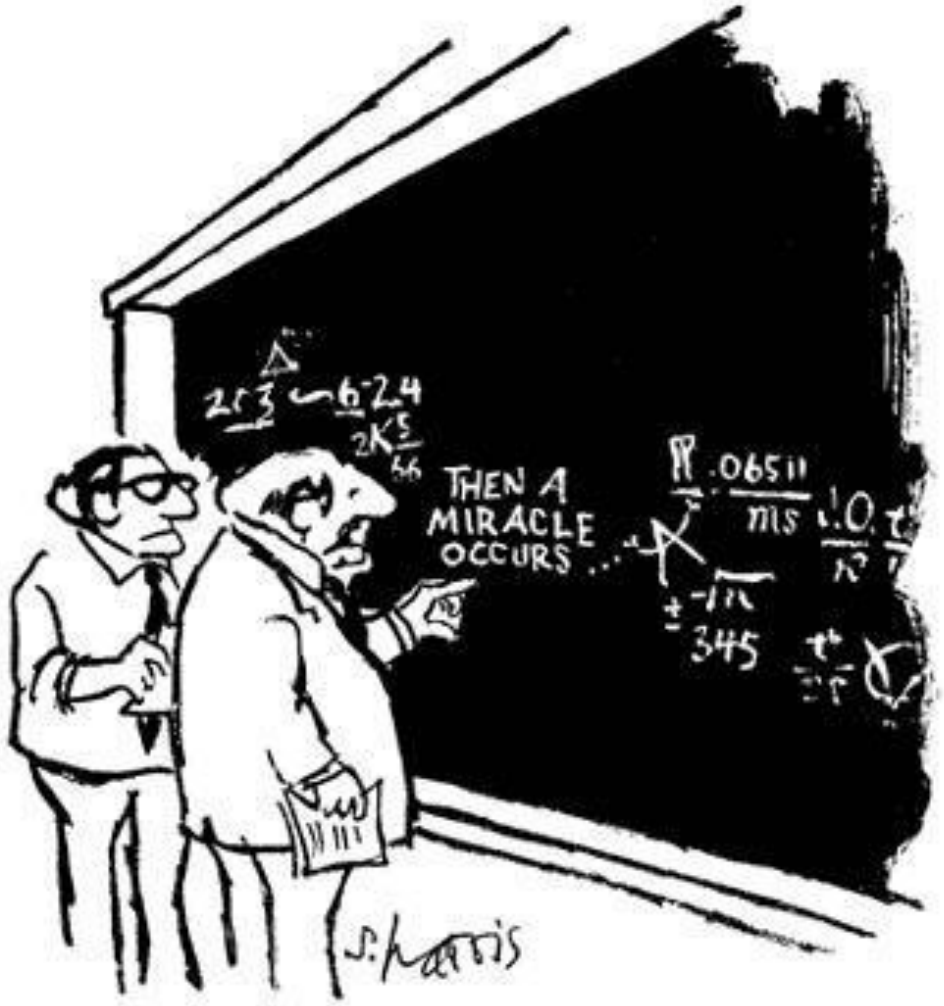
No stimulus! Just **constraints**

- also expressed as properties
- **assumed** rather than asserted
- "Given that signal C never goes high…"



© 2019 Arm Limited

How Formal Proof Tools Work



"I think you should be more explicit here in step two."

© 2019 Arm Limited

The Real World

This sounds amazing! Do we always get full proofs?

Reality is (sadly) not so rosy…

- Properties can be **undetermined**

But it’s not all or nothing

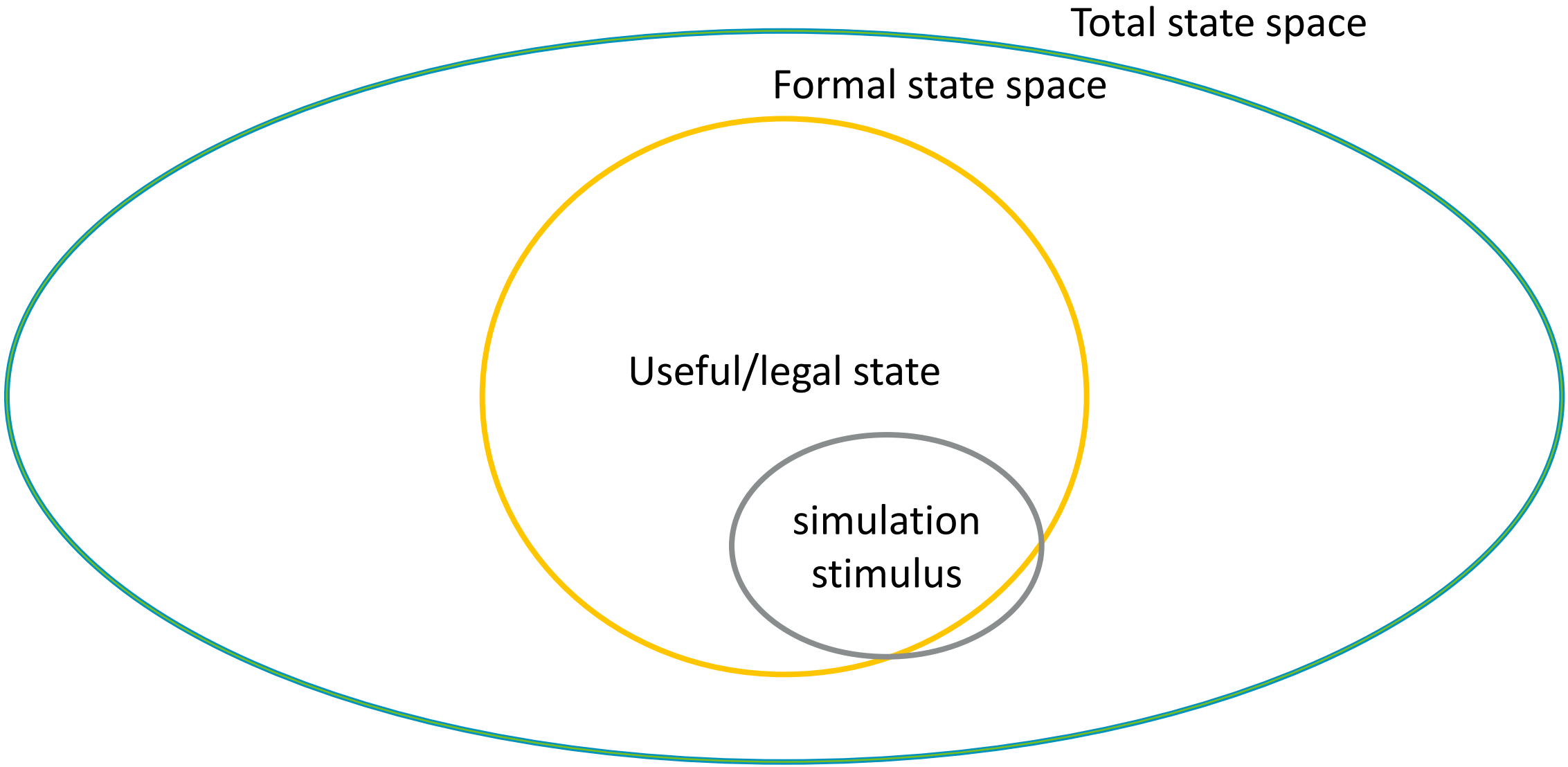
- We can get **bounded proofs** (more later)
- Formal is good at finding **bugs**
- **Counterexample** trace is shortest possible
- If we’ve run a property for a long time and found no counterexample this **increases confidence**

We have plenty of fancy tricks up our sleeve…

- State abstractions, abstract functions, invariants, oracles

© 2019 Arm Limited

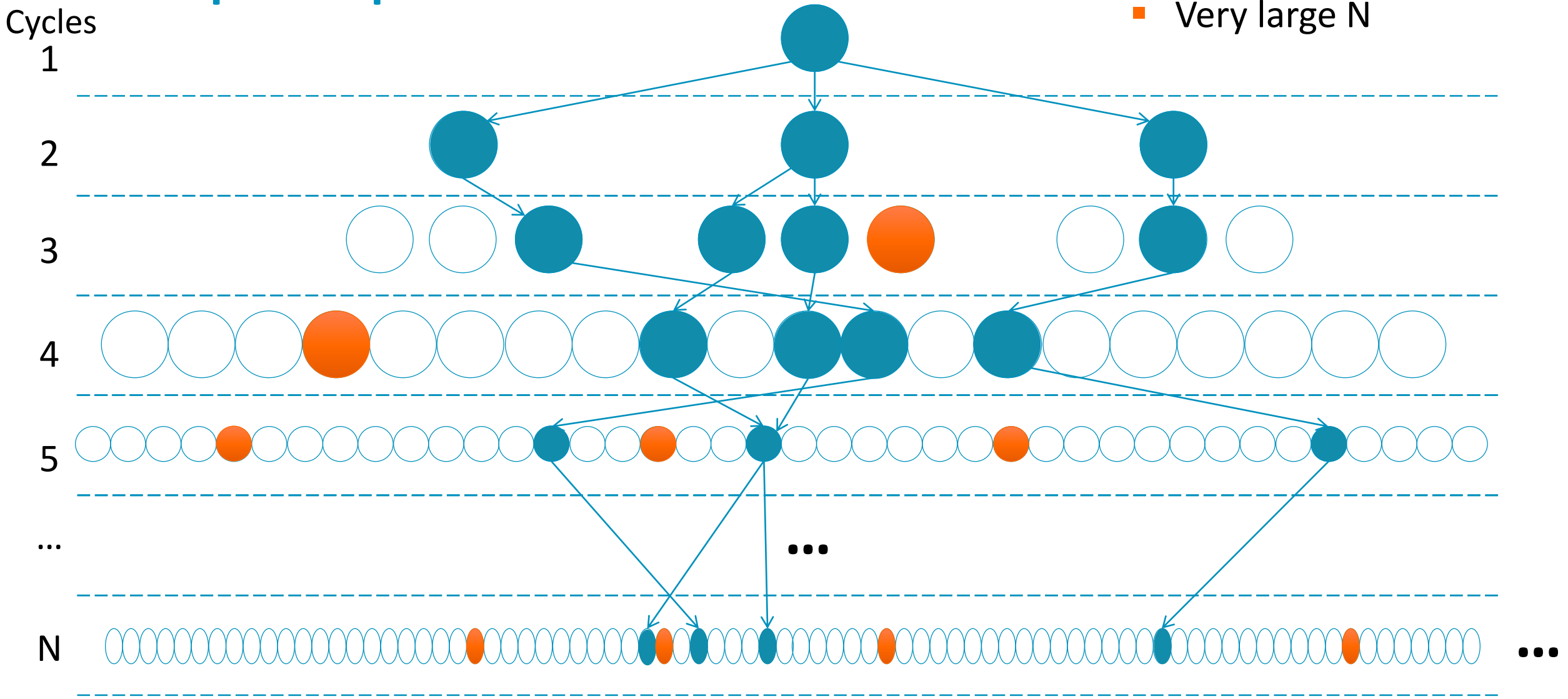
State Space Exploration



© 2019 Arm Limited
arm

State Space Exploration – Simulation

- "Depth-first" search
- Very large N

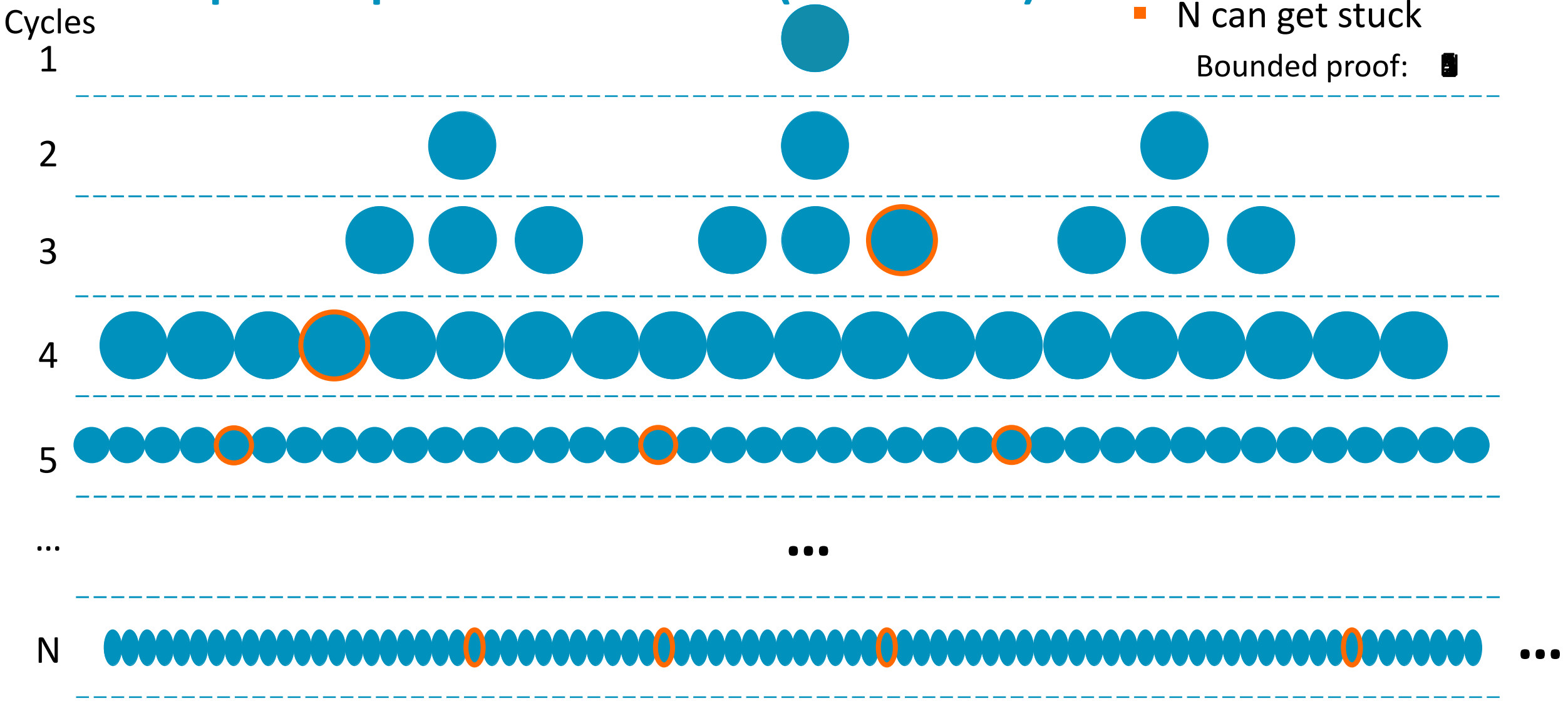


© 2019 Arm Limited
arm

State Space Exploration – Formal (exhaustive)

- "Breadth-first" search
- N can get stuck

Bounded proof: [redacted]



© 2019 Arm Limited
arm

State Space Exploration – Formal (exhaustive)

Some important caveats:

Covers state space exhaustively **for a given property**

- Can only find bugs if property can find them

Proof tool might get “stuck” at cycle N

- What if there’s a bug at cycle N+1?
- Simulation gets deeper quicker

Bugs can take a very long time to find

- 48 hours CPU time not unusual for corner cases

How long to run for when we can’t find a proof?

- Cycle depth? How deep is ‘deep enough’?
- CPU time? How long is ‘long enough’?

© 2019 Arm Limited

Complementary State Space Exploration

Simulation

Can miss cases due to non-exhaustive search

Only as good at finding bugs as the checkers you write (can use properties)

State exploration limited by the stimulus you write

Coverage is really hard to do well

Formal

Can miss deep cases due to getting ‘stuck’ with N < bug_depth

Only as good at finding bugs as the properties you write

Explores all states until you write constraints to say not to

Coverage is really hard to do well

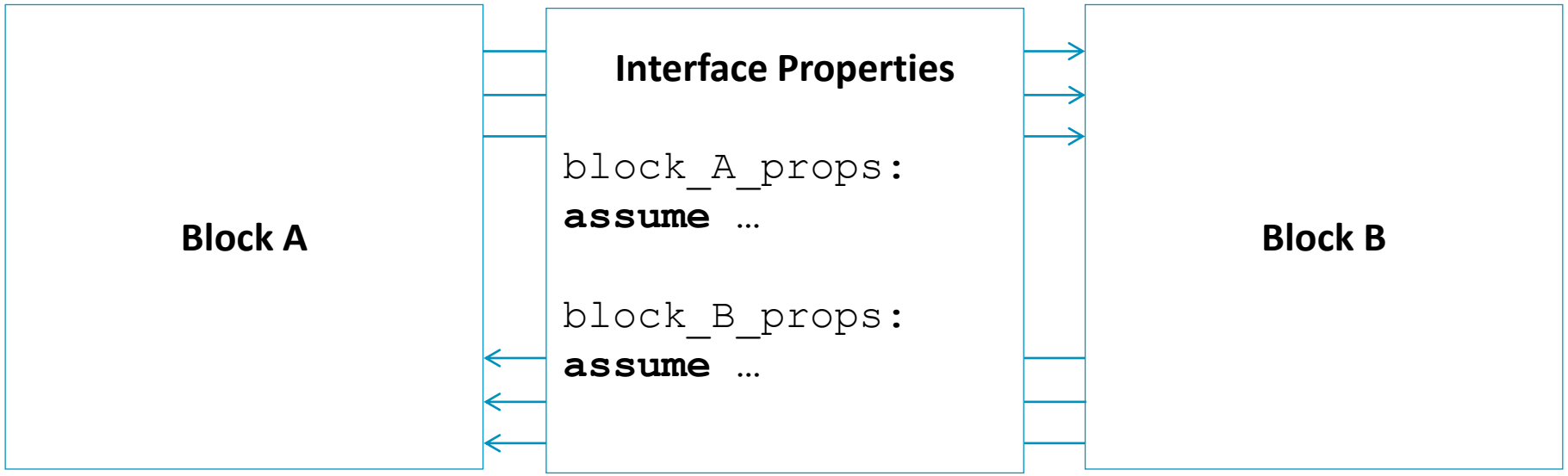
Advantages and Disadvantages on both sides → **Complementary approach**

© 2019 Arm Limited
arm

Interface Specifications and abstractions

Want to run formal on Block A embedded properties…

… but there's loads of logic in Block B → Loads more state space for tool to explore…



… but we still need the interface signals to behave correctly, e.g. a request from Block A still needs to be honoured…

… so let's just get rid of Block B and treat it as a **black box**…

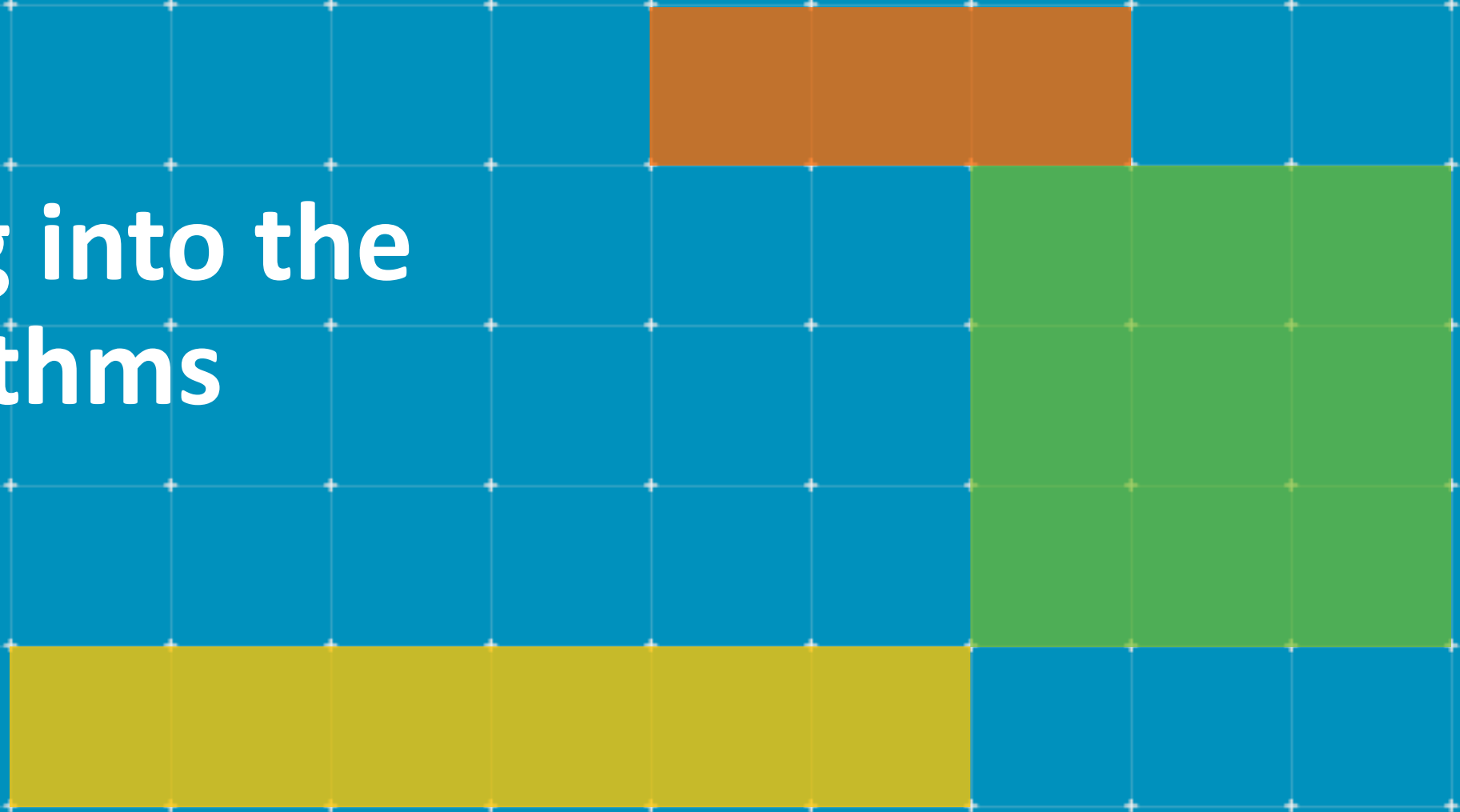
© 2019 Arm Limited
arm

Other abstractions

- “Cut” signals which have a large and complex fanin (driving logic), and which are not useful for the rest of the verification
- Black-box modules which are bad for formal verification: multiplers, dividers, encryption, etc.
- Reduce internal structures to make reaching corner cases easier: FIFO depth, counters, etc.
- Allow the formal analysis to start for non-reset states so that “filling” the structures does not consume cycles: cache RAMs
- Change arbitration policies to allow using some elements more directly
- Consider only a subset of the design when structure are symmetrical

© 2019 Arm Limited

Diving into the algorithms



© 2019 Arm Limited

arm

Formal verification techniques

Use a mathematical reasoning in order to solve problems

For hardware verification, *usually* decomposed at the bit-level and using rules of the propositional logic:

- Distributivity, commutativity, transitivity
- DeMorgan law
- Etc.

Main FV techniques are:

- Theorem proving and checking
- Symbolic simulation
- Model checking

© 2019 Arm Limited

Reduced Ordered Binary Decision Diagrams

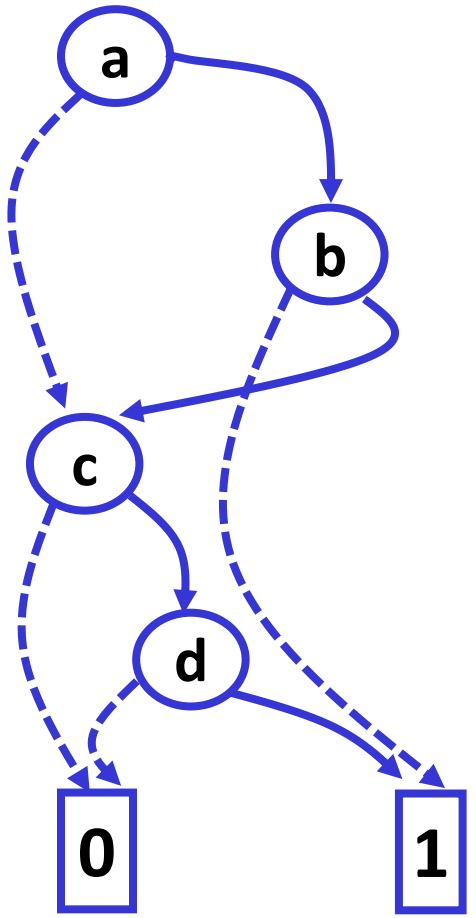
- Invented by Bryant in 1986
- Allow to represent a Boolean formula as a compact and canonical graph, based on the Shannon expansion *Sexp*:

$$Sexp\big(f(a,b,c,\dots)\big) =$$

$$a\ \&\ (Sexp(f(1,b,c,\dots)))\ \mid\ \sim a\ \&\ (Sexp(f(0,b,c,\dots)))$$

- Simple algorithms, in linear time and space complexity to combine BDDs. Example BDD_or = APPLY(OR, BDD_a, BDD_b)
- If the formula is always false, its BDD is the single node **0**
- If it is always true, it is **1**
- Otherwise, the BDD can be traversed to get the variable values leading to the formula being false.

a & (~b) | c & d

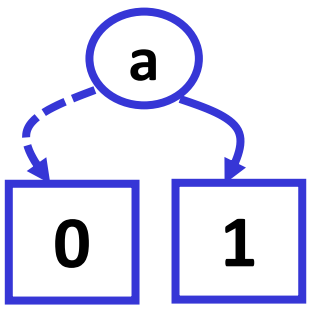


© 2019 Arm Limited
arm

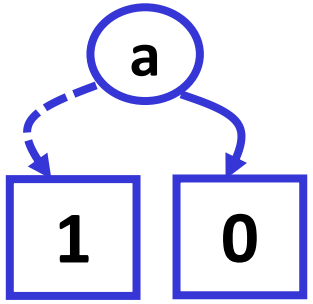
BDD construction example

$(a \,\&\, (\sim a)) \,\&\, b$ ==0

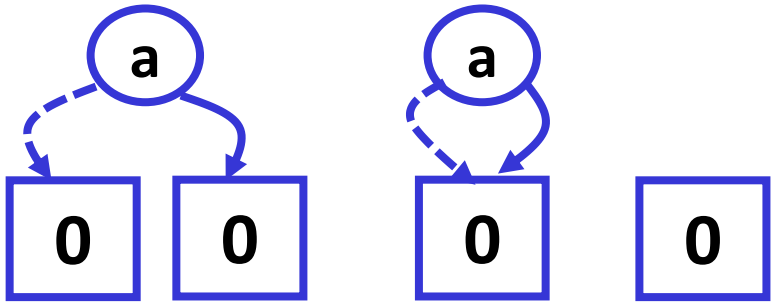
a



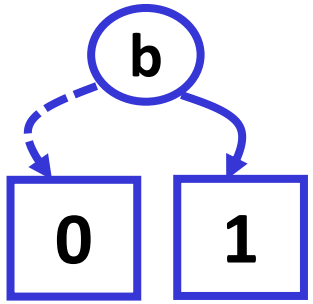
~a



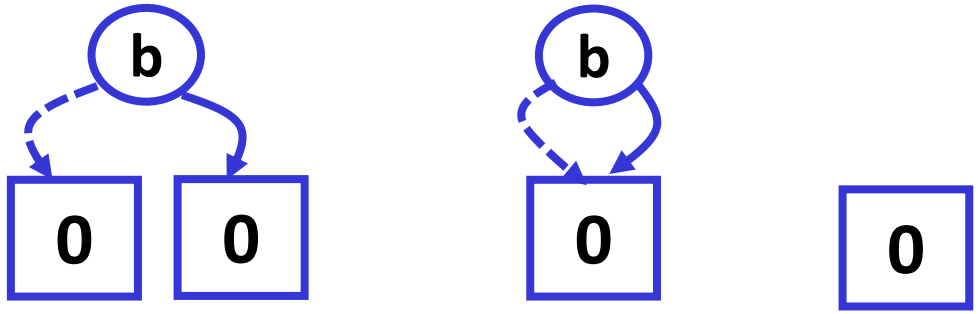
a & ~a



b



(a & (~a)) & b



© 2019 Arm Limited

arm

Usage of BDDs for formal verification... And others

- Build a single BDD representing the design logic AND-ed with the assertion to prove
- If it is **1**, the assertion is always true
- Otherwise a path leading to **0** gives input values that violate the assertion
- Huge Boolean formulas can be represented with small BDDs if the variable ordering is good
- But not all. E.g. multipliers and dividers

- BDDs are very efficient for combinational verification such as Logical Equivalence Checking
- Many improvements have been added (cache policy, dynamic reordering,...)
- BDDs are also used for logic minimization and synthesis

© 2019 Arm Limited
arm

Model Checking - principles

1. Build the **reachable** state space
2. Express an assertion using temporal logic: Boolean logic + extra operators: *all states*, *next state, eventually*, etc.
3. Check whether one of the reachable states violates the assertion. If none, then the assertion is proven

The state space can be built by

- forward image computation from the reset, until a fix point is reached
- Or from a state violating the assertion and backward to the reset state, if possible

© 2019 Arm Limited

Explicit state model checking

Invented by different research teams in early 80s. Turing prize for Clarke, Emerson, Sifakis in 2007

Automatically reason on a design and temporal assertions

Each state is explicit so the computer must be able to count them

A super set of the reachable state space has $2^{nb\_registers}$ elements

- For the Arm Cortex® A75 CPU, that would be 10^{133000} states
- If representing each state by a single atom, it would require 1511 universes

So, usable only to reason about very small state machines

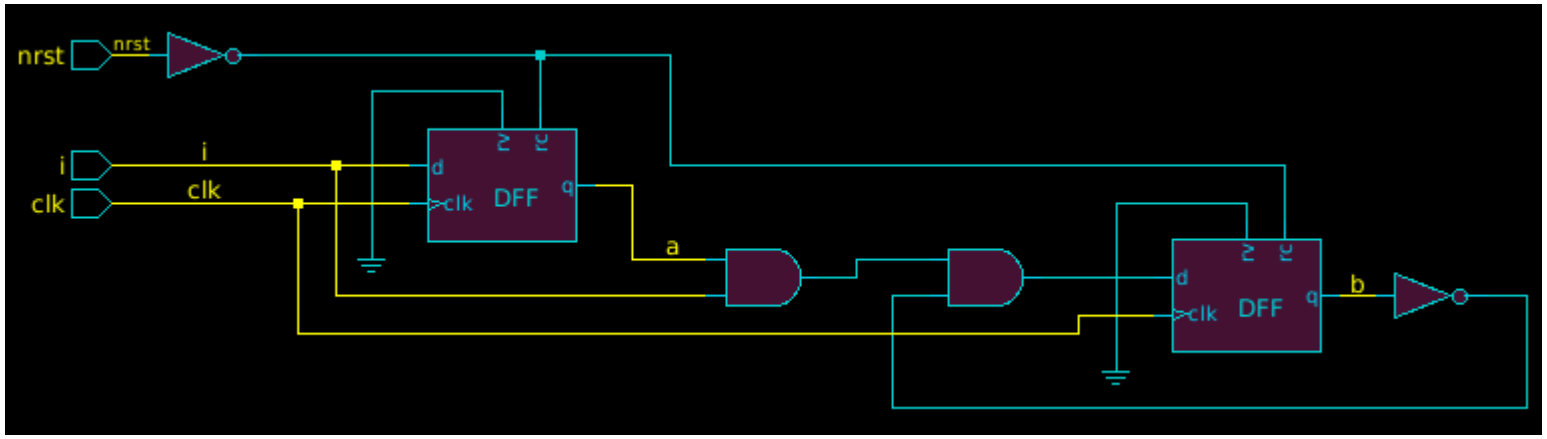


Image function:

a(0) = 0
b(0) = 0
…
a(1) = i(0)
b(1) = 0
…
a(n) = i(n-1)
b(n) = i(n-1) & a(n-1) & ~b(n-1)

Reachable state space:

	a = 0	a = 1
b = 0	✓	✓
b = 1	✓	✓

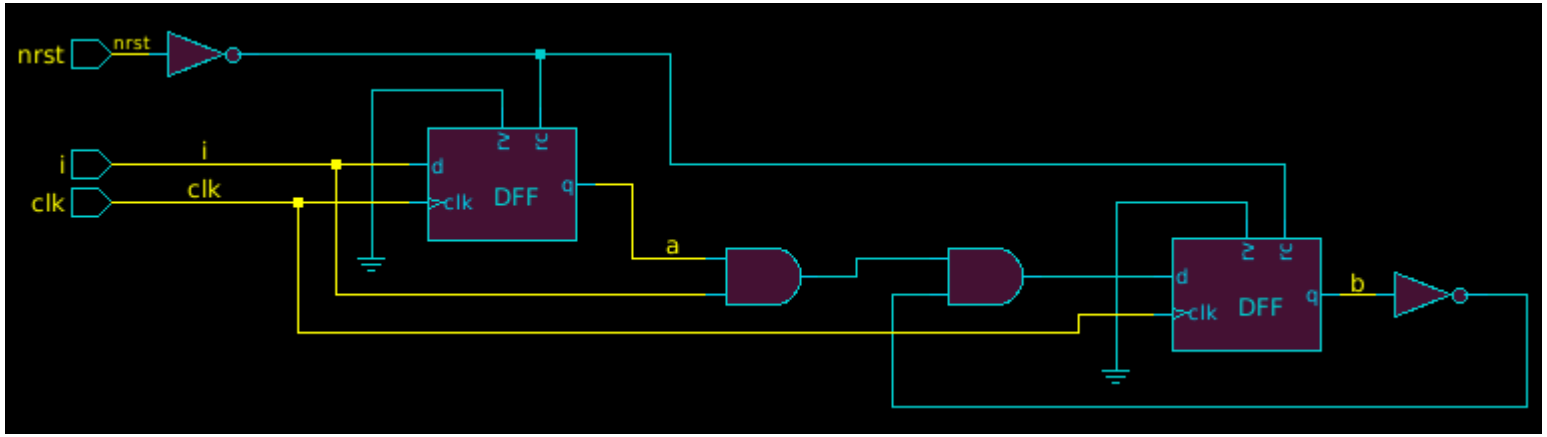
arm

Implicit state (symbolic) model checking

This was a revolution in model checking!

Clarke and McMillan in 90s

Instead of building the set of all reachable states, build a function with all registers as arguments, which is true if and only if it corresponds to a reachable state



Reachable state space:

$$\boldsymbol{R(a,b)} = \sim(\sim\boldsymbol{a}\ \&\ \boldsymbol{b}) = \boldsymbol{a}\ |\ \sim\boldsymbol{b}$$

© 2019 Arm Limited
arm

Symbolic model checking with BDDs

The R function may be huge...

...Here come the BDDs back: they allow to represent in a compact form the R function, especially for control-oriented designs

BDD functions can be used to reason about the state space

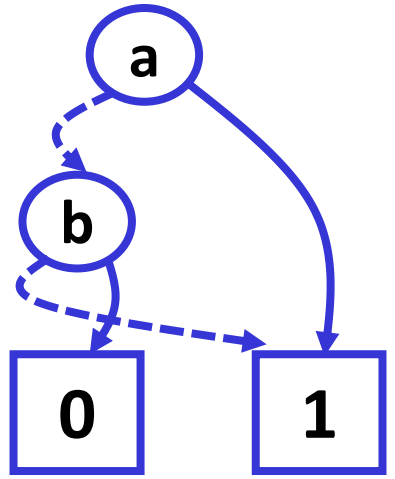
- Example: can a and b be high at the same cycle?
- Example: assert (b |=> ~b)

Liveness properties can also be checked

- Example: assert (b |=> s_enventually b)

Reachable state space:

$$\boldsymbol{R(a,b) = a \mid \sim b}$$



© 2019 Arm Limited
arm

After BDDs

Formal verification gained acceptance thanks to the development of symbolic model checking with BDDs

But hardware designers went faster and BDDs are not capable for the formal verification of modern designs anymore

BDDs still have great capabilities; you are using them without knowing it (synthesis tools, Facebook, biology, etc.) and they may have an interest for your home made tools (free and good BDD libraries are available)

Best results in formal verification are now obtained with SAT solvers

© 2019 Arm Limited

Boolean SATisfiability problem

Is a given propositional formula satisfiable?

$$(\sim x_1 \mid x_2) \,\&\, (\sim x_2 \mid x_3 \mid x_4) \,\&\, (\sim x_1 \mid x_3 \mid \sim x_5)$$

Satisfiable: there exists an assignment for the Boolean variables such that the formula holds.

- And by the way, what is the assignment!

Literal: x , $\sim x$

Clause: $\sim x_2 \mid x_3 \mid x_4$

Conjunctive Normal Form: $C_1 \,\&\, C_2 \,\&\, C_3$

© 2019 Arm Limited

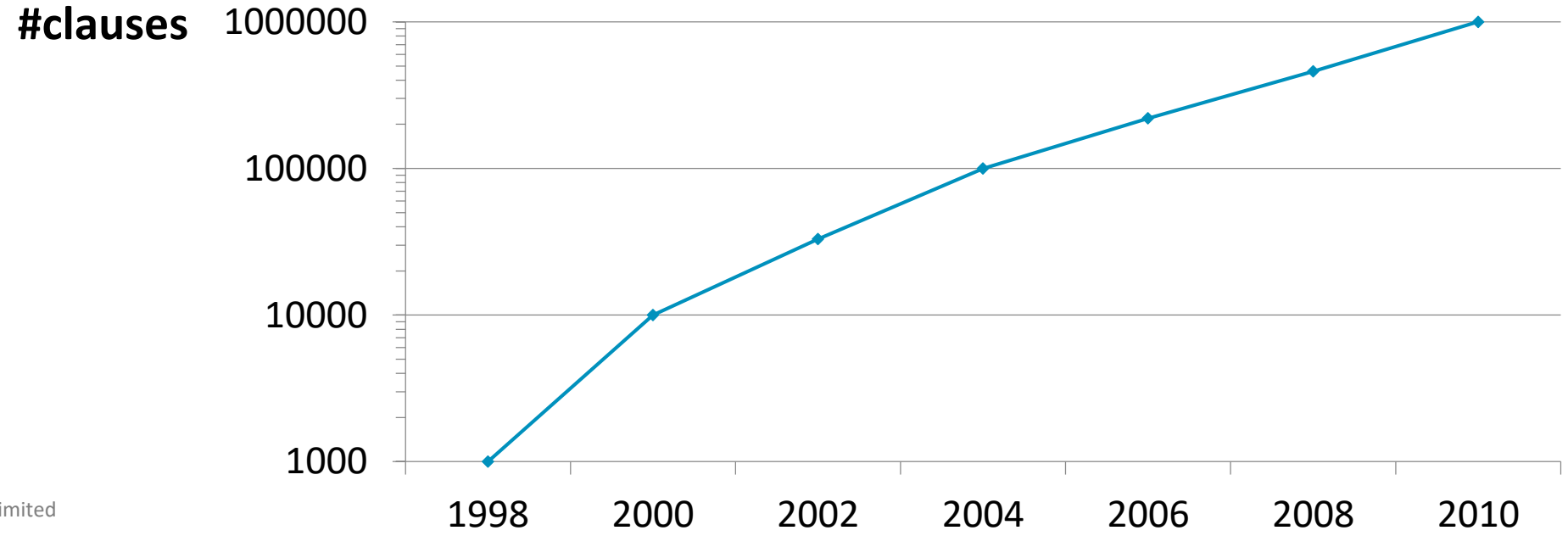
SAT complexity

k-SAT means clauses contain at most k literals.

2-SAT is in P. $(\sim x_1 \mid x_2) \,\&\, (x_3) \,\&\, (x_3 \mid \sim x_5)$

3-SAT is the first NP-complete problem. $(\sim x_1 \mid x_2) \,\&\, (x_3) \,\&\, (x_1 \mid x_3 \mid \sim x_5)$

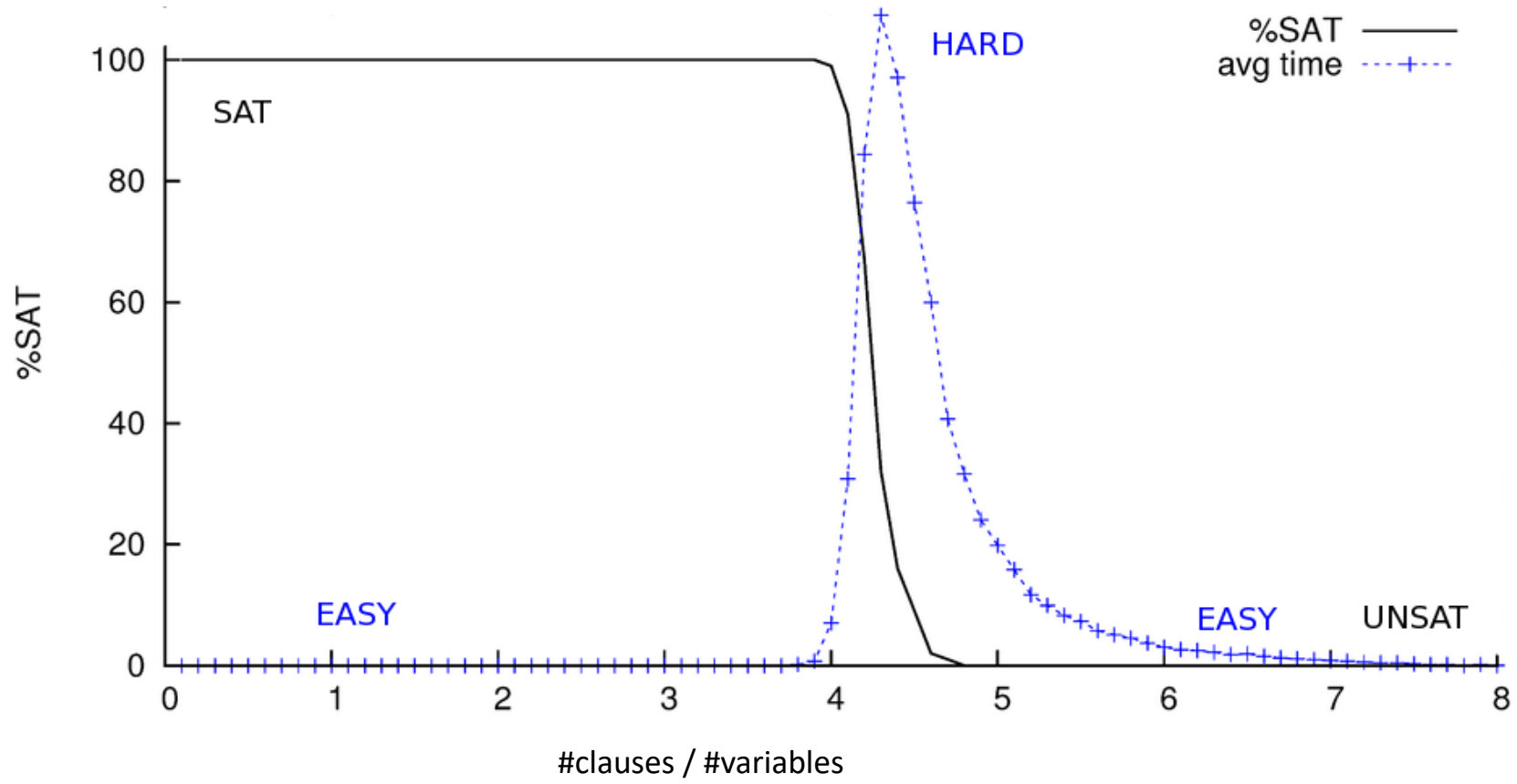
However SAT solvers are very efficient in practice.



© 2019 Arm Limited

SAT complexity – Random instances

Phase transition for random instances w.r.t. clauses over variables ratio.

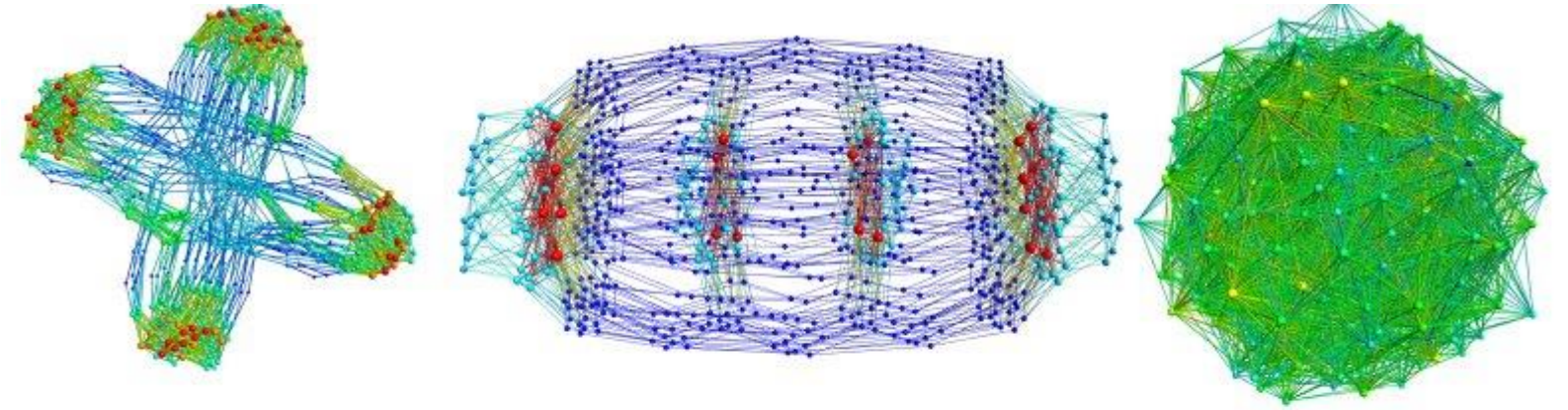


© 2019 Arm Limited
arm

SAT complexity – Industrial instances

Community graphs

- Variables are vertices connected by an edge if they appear in a same clause.
- *Tend* to show that hard random instances have few weakly connected clusters compared to industrial instances.

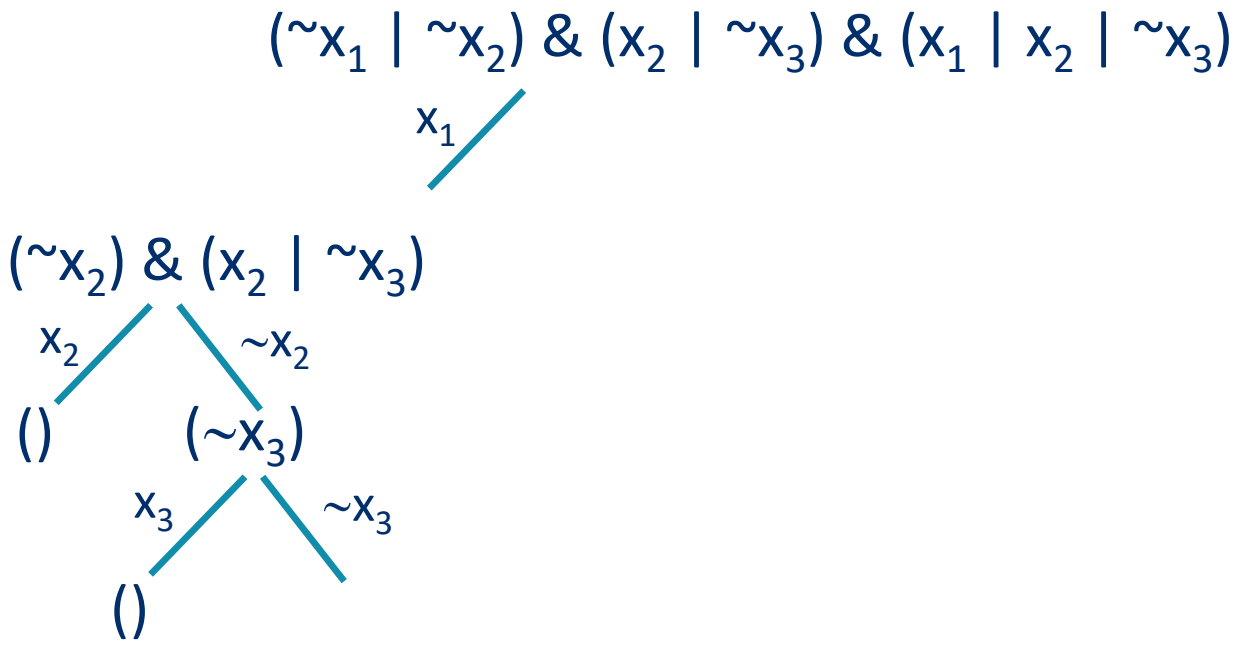


© 2019 Arm Limited

SAT solvers

Davis–Putnam–Logemann–Loveland (DPLL) algorithm (1962)

A complete method based on backtracking and a few deduction rules



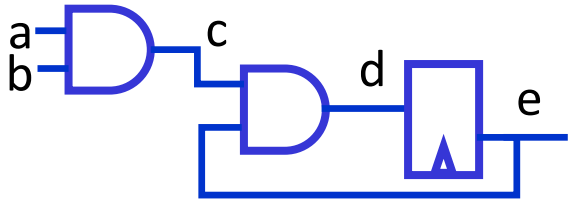
© 2019 Arm Limited
arm

SAT encoding of hardware circuits

Combinatorial logic can be written as a CNF formula (Tseytin, 1968)

- Each simple gate has a CNF formula. $c = a \,\&\, b \equiv (\sim c \mid a) \,\&\, (\sim c \mid b) \,\&\, (\sim a \mid \sim b \mid c)$
- Circuit formula is the conjunction of each gate formula.

Sequential logic is modeled with a transition relation:

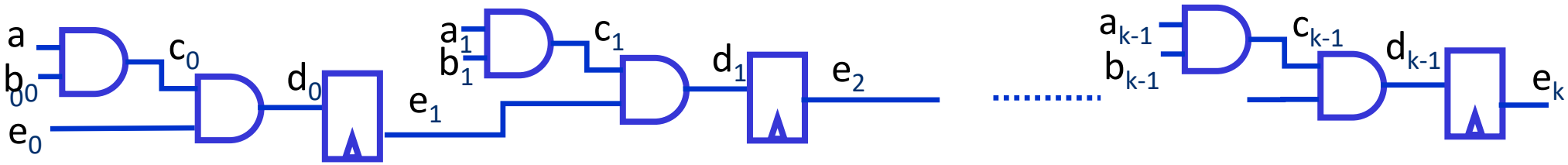


$$T(s_{n-1}, s_n) = (\sim c \mid a) \,\&\, (\sim c \mid b) \,\&\, (\sim a \mid \sim b \mid c) \,\&\, (\sim d \mid c) \,\&\, (\sim d \mid e_{n-1}) \,\&\, (\sim c \mid \sim e_{n-1} \mid d) \,\&\, (\sim d \mid e_n) \,\&\, (\sim e_n \mid d)$$

© 2019 Arm Limited

SAT encoding of hardware circuits

Model can be unfolded k times to represent k cycles.



$I(s_0) \& T(s_0, s_1) \& T(s_1, s_2) \& \ldots \& T(s_{k-1}, s_k)$

Each time the model is unfolded, the formula grows linearly with the size of the circuit, however possible assignments grow exponentially.

A trace in such a model is the value of the inputs at each cycle, that is a sequence $\{a_0, b_0, e_0\}, \{a_1, b_1\} \ldots \{a_{k-1}, b_{k-1}\}$ that satisfies the formula.

© 2019 Arm Limited

Bounded Model Checking (BMC)

A bug finding technique: Is there a counter-example of length k to property P?

Assuming a safety property:

$I(s_0)$ & $\sim P(s_0)$

$I(s_0)$ & $T(s_0,s_1)$ & $\sim P(s_1)$

$I(s_0)$ & $T(s_0,s_1)$ & $T(s_1,s_2)$ & $\sim P(s_2)$

... i=1...k $I(s_0)$ & $\&T(s_{i-1},s_i)$ & $\sim P(s_k)$

© 2019 Arm Limited

But wait, isn't formal about proofs?

BMC is complete in theory…

- Completeness threshold is the minimal k that ensures we have explored all reachable states.

…but limited to falsification in practice.

- Finding the completeness threshold is as hard as model checking itself.
- Use over-approximations:
  - Maximum number of states ≈ 2^nb of registers
  - Diameter = the longest of the shortest paths to any reachable state from the initial state
  - Recurrence diameter = the longest loop-free path

Unbounded SAT model checking (UMC)

- k-induction, interpolation, CEX-based abstraction, proof-based abstraction,…

© 2019 Arm Limited

SAT summary

SAT solving is now the main workhorse of formal verification.

- Especially for bounded model checking.
- BDDs still have a place for proving the absence of errors and for liveness proofs.

Other application of SAT solvers

- Error localization
- Planning
- Optimization

Higher-level solvers

- Satisfiability Modulo Theory
- Constraint Programming

© 2019 Arm Limited

Some applications

- Embedded assertions
- Coverage
- Sequential equivalence checking
- Security verification
- ISA-formal

© 2019 Arm Limited

arm

Embedded assertions

- Check designers' assertions
  - Written for both simulation and formal
- Several flows
  - Design bring-up: over-constrained, to exercise simple behaviours early
  - Regression: cluster friendly for thousands of properties. Maximize parallelism and balanced engine selection
  - Soak: never ending runs targeting bug hunting

© 2019 Arm Limited

Coverage

- Allows various analyses
  - Reports dead code
  - Generate waivers for simulation code coverage and vice versa
  - Can help filling the latest % in simulation code coverage
  - Increases confidence in the formal environment not over-constraining the design
  - Shows parts of the design weakly covered by assertions
- But some Cover Items may be statically covered, but not used to get proofs
- Towards a formal coverage metric: proof core
  - Reports Cover Items explored by abstraction engines
  - Approximates progress of formal verification effort

© 2019 Arm Limited

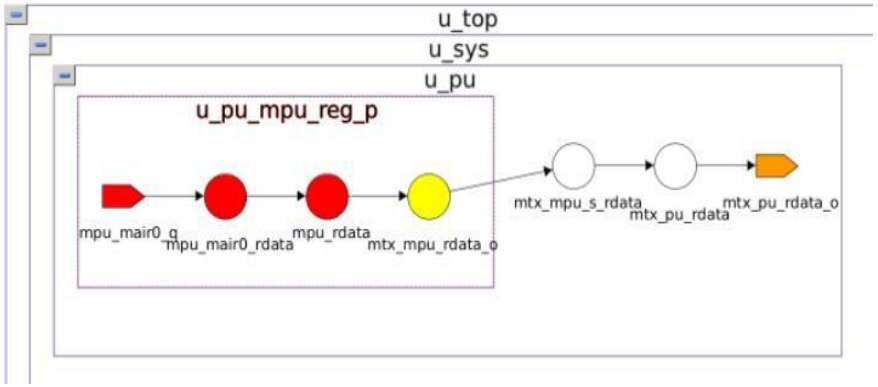
Sequential Equivalence Checking

- Prove that two different circuit descriptions exhibit exactly the same functional behavior:
  - Same design with and without a given feature
  - Rewriting, optimizations, retiming, ECO
  - Mid-level and Top-level clock-gatings
- Can only be handled by formal methods
- Advanced techniques used to get exhaustive proofs
- Now investing on C-RTL equivalence checking also

© 2019 Arm Limited

Security

- Used to check designs for security leaks:
  - Currently, only for secure core (close to M-class processor)
- How does it work?
  - The principle is to prove that no secure data can be seen outside a secure zone
    - Checks for unwanted paths from secure to non secure areas. Example: DRM keys
- Latest leaks showed that security for A-class processor is a new sensitive and challenging area



© 2019 Arm Limited
arm

ISA Formal: the Holy Grail for formal verification of CPUs

- Instruction Set Architecture-level
- Used to determine whether a given instruction retires having performed all its architected behaviour.
- Uses the Architecture Explorer (ArchEx) tool to auto generate Verilog from the ARM ARM pseudocode.

AUTOGENERATED
logic to predict
correct next state

© 2019 Arm Limited
arm

To conclude

- Formal verification interests and ROI are growing up every day

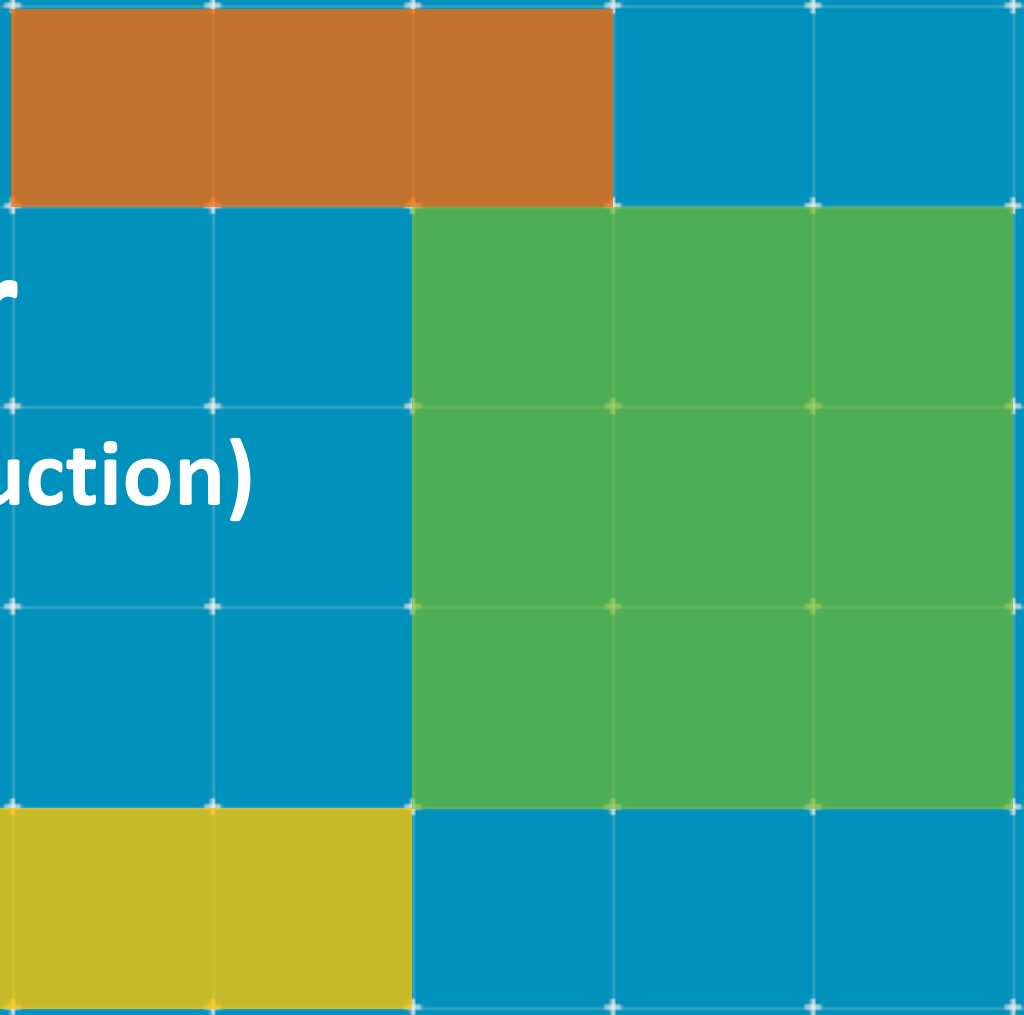
- It complements other verification methodologies
  - Using both methods ensure well-rounded verification
- Formal verification is agile and great for “design-bringup”
- And more and more small or derivative projects use formal as their primary methodology

Formal verification is not boring mathematics!

It’s an innovative and efficient technique that is still improving

© 2019 Arm Limited
arm

Formal methods for software (a short introduction)



© 2019 Arm Limited

arm

Hardware vs Software

- Hardware designs are easier to verify than softwares:
  - The semantics of the hardware description languages are well defined
  - If respecting some simple coding guidelines, all designs can be converted into a giant Mealy machine FSM at the bit level, which is what the formal tools want as input
  - Software languages vary a lot:
    - Pure functional languages have a well defined semantics (Scheme, Lisp, Haskell,…)
    - Others have not (C, Python,…), but can still be formally verified
    - Programming features such as pointers and Object-oriented make things much more difficult
  - Software programs have a lot larger input and state spaces than hardware designs
- Having a bug-free hardware is more important than a bug-free software:
  - software can be fixed after production (new versions),
  - hardware cannot – usually
- Hence, formal methods are more widely used in the hardware design industry than in the software industry

© 2019 Arm Limited

Formal methods for software

- However, fundamental research works applied on software:
  - Hoare logic (precondition, postcondition, invariants)
  - Theorem provers (e.g. ACL2, HOL, Coq)
  - Real-time languages and tools (e.g. Esterel, Lustre)
- Good model checking tools exist for software:
  - CBMC (Oxford University), Z3 (Microsoft), etc.
- Some industrial tools exist:
  - Mostly related to specification and refinement:
    - A high-level specification is manually written
    - It is *validated* (not *verified*) using assertions and tools
    - It is refined, automatically or not, in one or several steps down to the real software
  - Scade (based on the Lustre language), B-method, UML,...

© 2019 Arm Limited

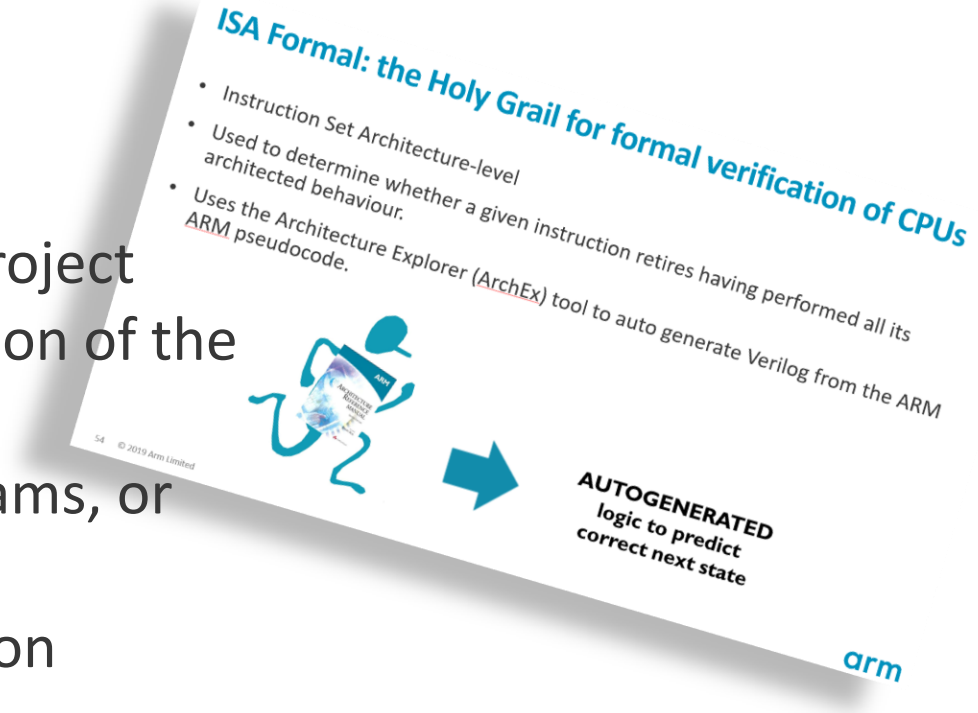
How YOU can use formal methods to verify your software?

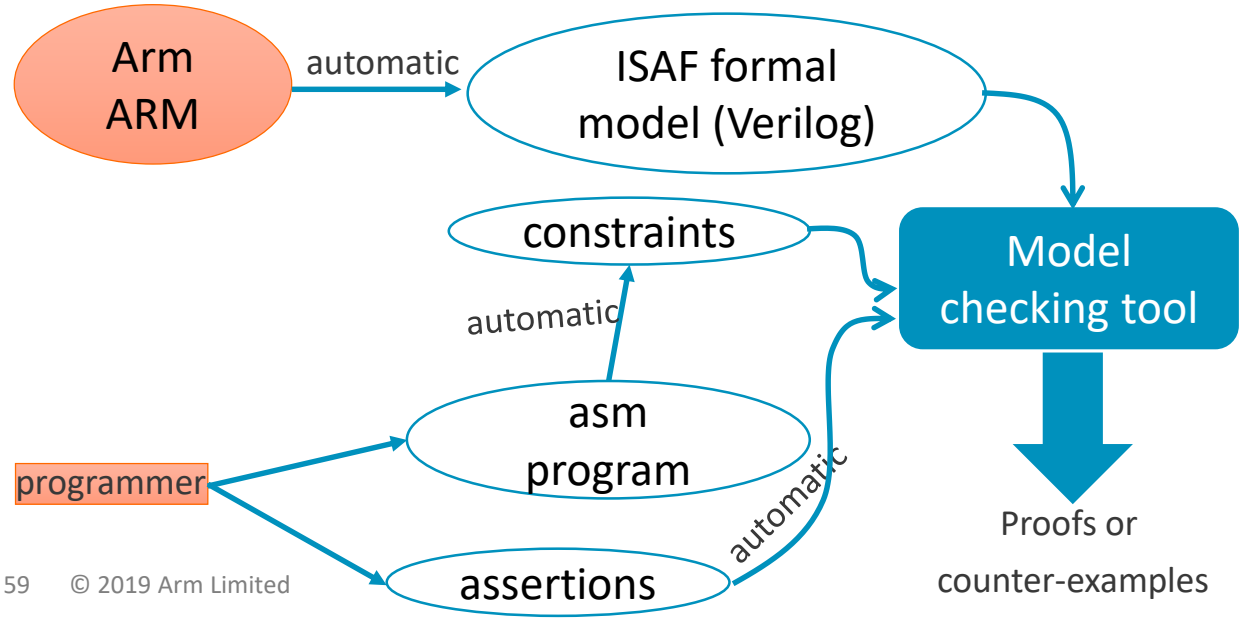
1. Choose the appropriate language
   - A functional language is probably the best choice
2. And use only a well defined subset of it
   - E.g. Scheme without set!
   - C without pointers
3. Use *clean* programming
   - No global variable
   - No break statement
4. Write pre-conditions and post-conditions in functions, and invariants in loops
5. Prove (using a tool, or manually) that your post-conditions and invariants hold
6. You're done!

© 2019 Arm Limited

A brief description of SWISAF

- SWISAF = “Software ISA-formal” is a toy internal project
- It uses the ISA-formal model as a formal specification of the Arm assembly language
- Allows to formally verify assembly language programs, or to compare 2 programs
- Uses a model checking tool for hardware verification






```
; init registers
ASSUME x3_init_value: x3' == 12345678
  MOV x1, #3
  MOV x2, x0
; start of loop
here: ADD x2, x2, #2
ASSERT loop_inv_cnt_not_done: x1 != 0
  SUB x1, x1, #1
ASSERT loop_inv_partial_result: x2 + x1 * 2 == x0' + 6
  CBNZ x1, here
ASSERT loop_exit_cnt_done: x1 == 0
  MOV x0, x2
ASSERT end_pc: pc == pc' + 24
ASSERT end_result: x0 == x0' + 6
ASSERT x3_unchanged: x3 == 12345678
```


© 2019 Arm Limited
arm

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks

 © 2019 Arm Limited