

The background of the slide is a photograph of a wind farm on a beach. The wind turbines are silhouetted against a bright, hazy sky at sunset or sunrise. The ocean waves are visible in the foreground, and a few people can be seen walking on the beach. The overall tone is warm and serene.

arm

# Easy deadlock verification and debug with advanced formal

DAC'20 extended version

Laurent Ardit, Vincent Abikhattar – Arm Ltd., France  
Joe Hupcey, Jeremy Levitt – Mentor, A Siemens Business, USA

September 2020

# Author bios



**Laurent Ardit. Arm, CPU, Sophia-Antipolis, France**

20+ years experience in the semiconductor and EDA industry. Expert in formal verification, modeling, and high-level synthesis. Leading a CPU formal verification team. Ph.D. from Nice, France



**Vincent Abikhattar. Arm, CPU, Sophia-Antipolis, France**

Senior engineer in formal verification. Working on the next generation of Arm application CPUs, especially in the formal verification targeting the memory sub-system. M.sc from Grenoble, France



**Dr. Jeremy Levitt. Mentor, A Siemens Business; Fremont, CA office**

Principal Engineer, Formal Verification Group. Manages R&D for the Questa Formal product line, with 25+ years working on formal verification in EDA. Ph.D. in Electrical Engineering from Stanford University; B.A.Sc. in Eng. Science from the University of Toronto



**Joe Hupcey III. Mentor, A Siemens Business; Fremont, CA office**

Verification Product Technologist. Manages the Questa Formal product line. 15+ years in Product Management; Pre-MBA ASIC & FPGA D&V. Cornell University BSEE, MENG, MBA

This presentation is an illustration of a close cooperation between EDA and a semiconductor company.

Arm uses the Mentor QuestaFormal tool to verify CPUs. Deadlock checks being one of the most difficult task, we show here how an innovative tool feature helps to tackle it.

# Design deadlocks are critical and difficult to find

- The most difficult bugs to find in hardware designs are deadlocks, livelocks and QoS issues
- Traditional techniques to detect them in **simulation/emulation** are:
  - Add local watchdogs (e.g. FSM does not stay in state S for more than N cycles)
    - It is difficult to find the real N
    - They may find very localized issues, but not larger ones like livelocks
  - Add a global watchdog
    - Difficult to define the global “progress”
  - It is not exhaustive anyway!
- Traditional methods with **formal verification** are:
  - Proof of *liveness assertions* with the SystemVerilog semantics
  - Semi-formal bug-hunting techniques. Not mature yet, not exhaustive

# What's wrong with the formal verification of liveness asserts?

- *Safety assertions* are on the form “something bad must not happen”:  
`assert property (something |-> !bad_event)`
- *Liveness assertions* are “something good must always eventually happen”:  
`assert property (something |-> s_eventually good_event)`
- Often the liveness assertions fail in a formal proof: they check for *maybe-escapable deadlocks*
- *Fairness constraints* must be added:  
`assume property(s_eventually (trigger_for_good_event))`
- But this is a difficult task, and may be incorrectly done, so masking bugs
  - May not be able to verify the fairness constraints as liveness asserts on other blocks
  - High risk of incorrect circular reasoning when using the *assume/guarantee* technique



# New formal-based deadlock detection: perform 2 checks



## Maybe-escapable deadlock (LTL semantics, SVA):

*The koala has an escape route from the tree, but does not want to take it.*

*Adding the fairness constraint that the tree will eventually not provide food anymore may encourage him to move?*




## Unescapable deadlock (CTL semantics):

*The raccoon has no escape from the cage.  
Whatever happens in his environment, he is trapped!*



# New formal-based deadlock detection: combine results

- Each assertion has 2 results: maybe-escapable, and unescapable deadlock

	Proven as not maybe-escapable	Maybe-escapable
Proven as not unescapable	 No deadlock	 Found deadlock is escapable Must examine the escape event
Unescapable	-	 A real deadlock exists Probably a design bug

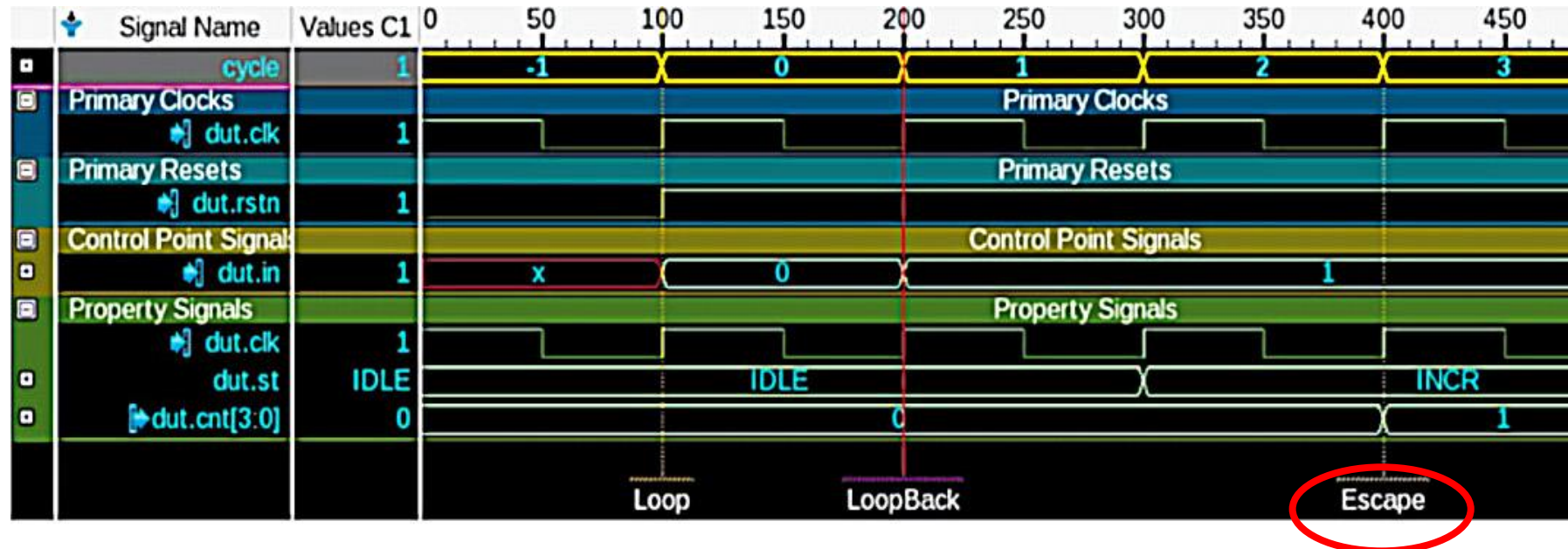
# New formal-based deadlock detection: undetermined cases

- However, formal can't always get a precise result

	<b>Proven as not maybe-escapable</b>	<b>Maybe-escapable</b>
<b>Undetermined</b>	No deadlock, except if incorrect fairness constraints	A maybe-escapable deadlock exists. Must debug

# Escapable deadlock: waveforms

New tool feature: an escapable deadlock result comes with a waveform showing the event which allows to exit from an otherwise infinite loop





# Escapable deadlock: what do we do?

Examine the waveform. Two cases:

1. Escape condition is not *interesting*. Add safety constraints to avoid them and rerun

E.g. warm reset, or ECC fatal error detection which puts the design in IDLE state

```
assume property (!warm_reset && !ecc_fatal_error)
```

2. Escape condition is valid (not a real deadlock). Add fairness constraints and rerun to ensure both checks pass

```
assume property (req |-> s_eventually ack)
```

This debug work is much simpler than the one with the traditional method looking only at maybe-escapable deadlocks.

# Unescapable deadlock: is there anything to do?



This is a real design bug

Open a new ticket assigned to design team

No need to ensure the failure is not due to missing fairness constraints

Having the extra information that it is not an escapable deadlock allows to reduce debug time a lot. No risk of adding unnecessary and incorrect fairness constraints

# Method applied to a large CPU in development (1)

- Instruction Fetch unit FSMs
  - Local FSMs are resilient to incorrect or unexpected environment behaviors
  - Maybe-escapable deadlocks are frequent, and their escape conditions are safe
  - A few results showed unescapable deadlocks
    - Some real design bugs, not found by any other method
    - Interesting issues with formal abstractions and their related constraints, not visible with a simple reachability analysis:  
`assert property (s_eventually(event))`  
is a much stronger check than  
`cover property (event)`
  - Proof time is a few minutes, with no overhead for also running the unescapable deadlock checks

# Method applied to a large CPU in development (2)

- L1 data cache arbiter
  - Mix of static and dynamic arbitration policies, with 6 requesters and optimized for performances
  - Liveness properties on the form

```
assert property (req_A |-> s_eventually(grant_A))
```
  - Maybe-escapable checks failed and would need lots of fairness constraints to model requester behavior
  - Unescapable checks helped to clarify specs, to push for more validation on requesters, and finally provided proofs
- Credit-based protocol
  - Can prove that no credit is lost
  - A few critical bugs found

arm

Thank You

Danke

Merci

谢谢

ありがとう

Gracias

Kiitos

감사합니다

धन्यवाद

شكراً

ধন্যবাদ

תודה





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](http://www.arm.com/company/policies/trademarks)