



JUNE 23-27, 2024

MOSCONE WEST CENTER
SAN FRANCISCO, CA, USA



Enhancing quality and reducing verification effort for RTL implementations against high-level C/C++ models using formal equivalence

Gaetano Raia ^[2], Gianluca Rigano ^[1], David Vincenzoni ^[1], Maurizio Martina ^[2]

[1] STMicroelectronics

[2] Politecnico di Torino



C versus RTL: current verification approach

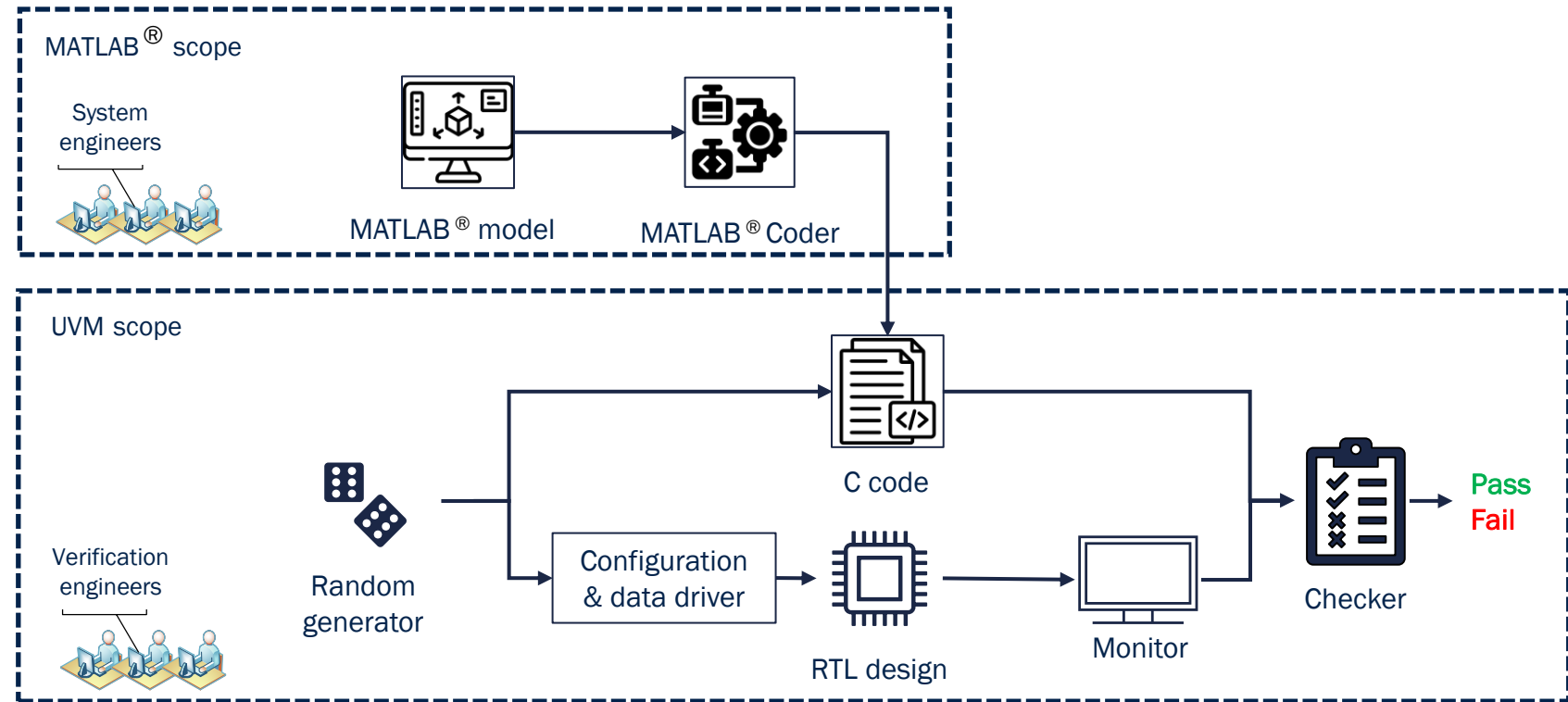
The UVM verification checks that, given the **same input values**, C and RTL models match, therefore they provide the **same output values** under the same input conditions

Generation of input patterns

Feeding the two models

Output checking process

Verification ends if **no error found** and reached **coverage**



The motivation – limits of UVM verification

However, **UVM dynamic simulations** exhibit some **limits** because of its inner aspects



Time-consuming

Environment
setup and tests
development
could require
weeks/months



Resource-intensive

Need of
implementing
reusable
verification
components



Prone to human error

Additional
verification efforts
while building the
testbench



Partially covering

Unfeasible
exhaustiveness
due to inputs and
state size

The main idea – formal equivalence verification

Formally verify that a **high-level C/C++ model** is equivalent to an **RTL design** by observing the output values



Exhaustive coverage

Mathematical-based approaches searching for CEX



Reduced verification time

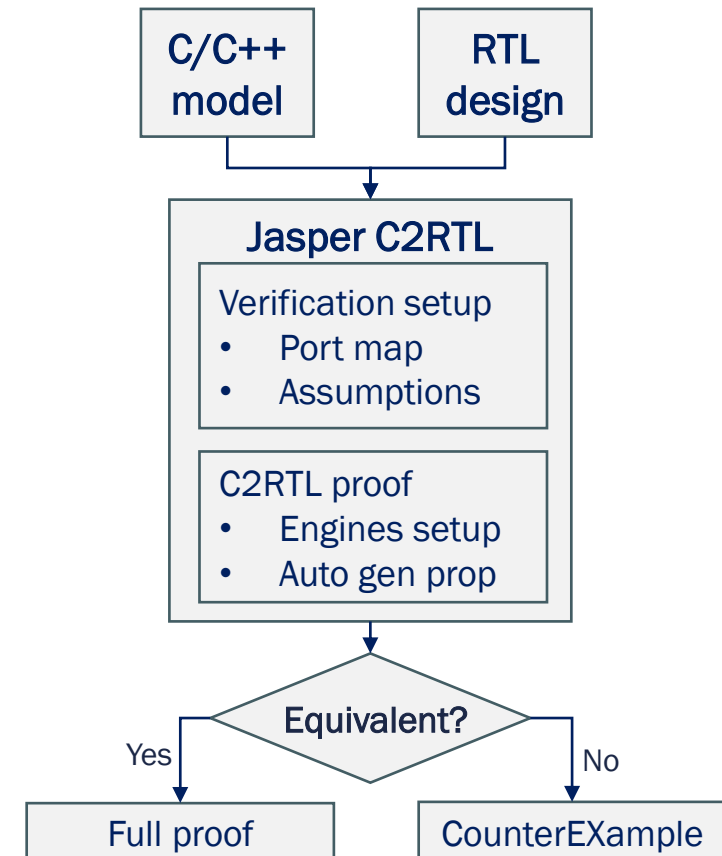
Verification setup requires port map and assumptions



Automatically generated assertions

The two models are said equivalent if providing the same outputs

Used formal verification tool:
Jasper C2RTL



The novelty and the challenges

The traditional use

Formal verification is used to compare two different **RTL designs** having the same behavior

The novelty

Recent improvements in formal engines have allowed to **extend the verification** of RTL designs with **high-level models**, enabling **early-design verification**

Challenges

Untimed C/C++
vs timed RTL

Complex cone of
influence (COI)

C2RTL born for pure
datapath circuits

Reconstruction of FSM-like datapath behavior

How to deal with an untimed C/C++ model (unable to keep past values) and a timed FSM-like RTL design?

Case study: fixed-point multiplier accumulator



Extension of C model interface

Extra dummy input signals to reconstruct the feedback



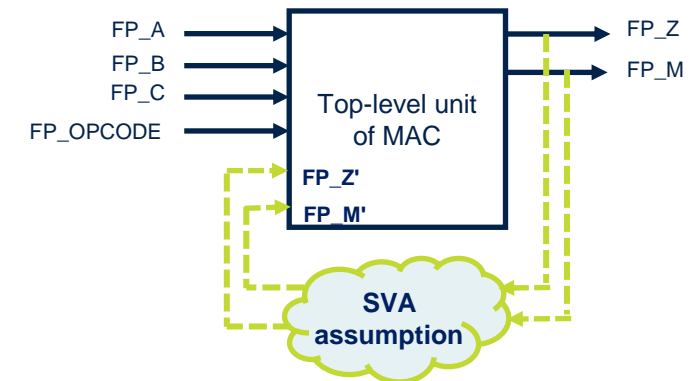
SVA assumptions

Force extra inputs to assume previous output values



Scalable bit width

Reduced bit width allows faster convergence



Snippet of code for the SVA assumptions

```
assume property -bound 1 (FP_M' == 0)
assume property -bound 1 (FP_Z' == 0)
assume property (##1 FP_M' == $past(FP_M))
assume property (##1 FP_Z' == $past(FP_Z))
```

Decomposition of complex cone of influence

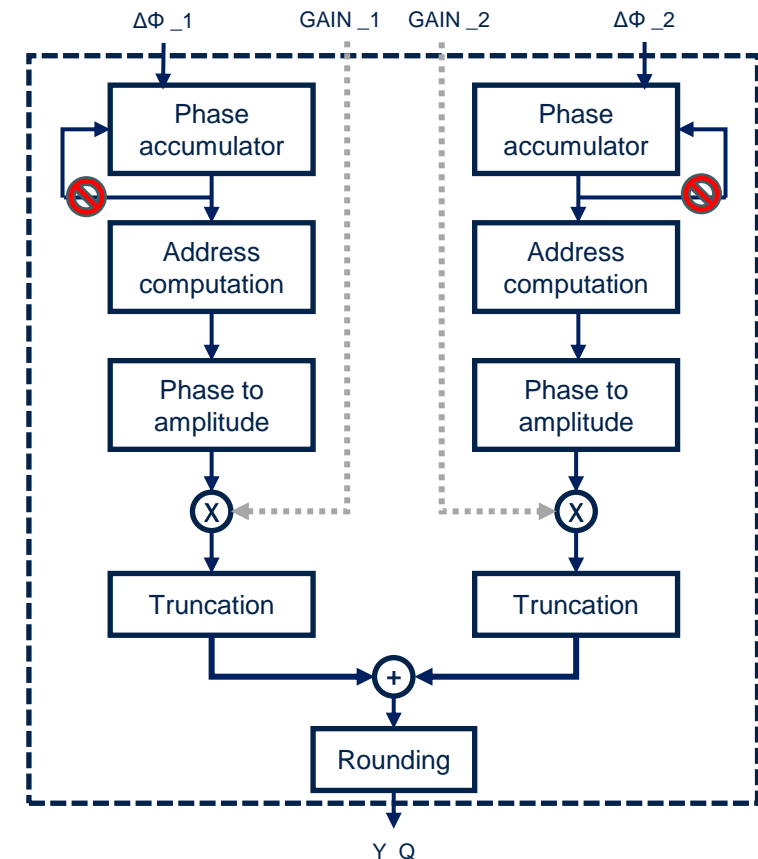
How to decompose complex cone of influence due to hard to prove automatically generated assertions?

1st

Overconstrained node

Phase accumulator value driven arbitrarily by the tool, overcoming feedback reconstruction

Case study: Tone Generator



Decomposition of complex cone of influence

How to decompose complex cone of influence due to hard to prove automatically generated assertions?

1st

2nd

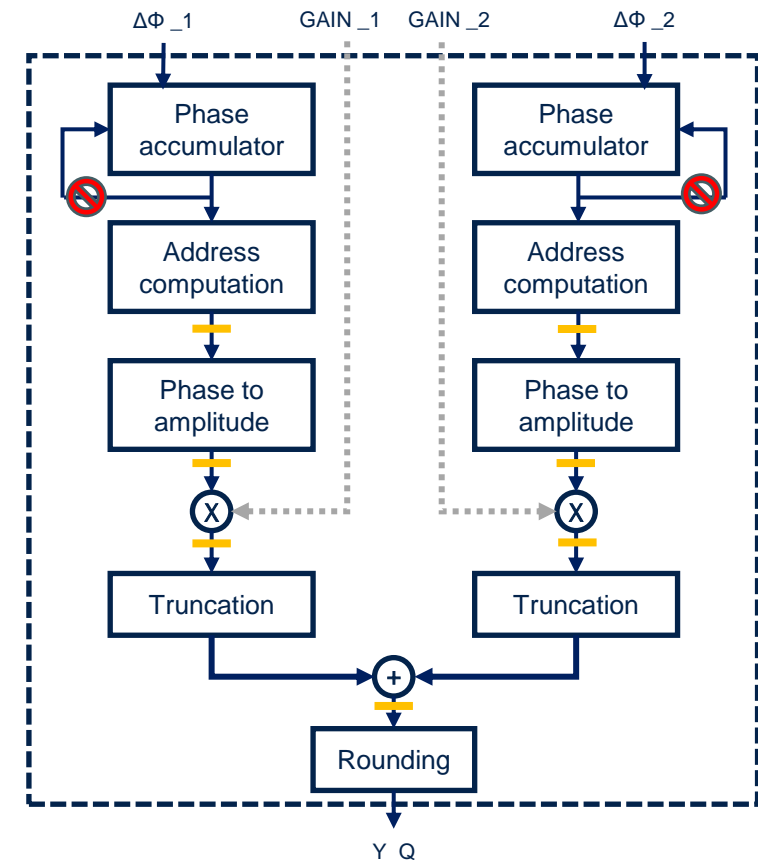
Overconstrained node

Phase accumulator value driven arbitrarily by the tool, overcoming feedback reconstruction

Intermediate equivalence points

Additional manually written **properties**, in intermediate points, help to **converge faster**

Case study: Tone Generator



Decomposition of complex cone of influence

How to decompose complex cone of influence due to hard to prove automatically generated assertions?

1st

2nd

3rd

Overconstrained node

Phase accumulator value driven arbitrarily by the tool, overcoming feedback reconstruction

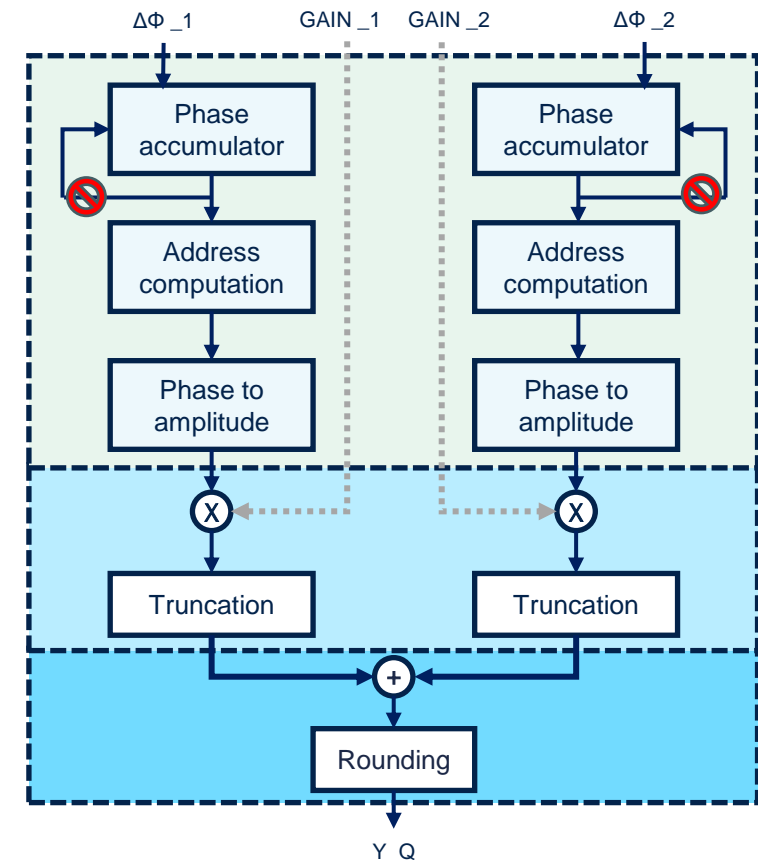
Intermediate equivalence points

Additional manually written properties, in intermediate points, help to converge faster

Assume guarantee

Proven assertions in intermediate points are used as assumptions for properties in cascade

Case study: Tone Generator



Jasper C2RTL using a MATLAB®-derived C model

How to handle **significant algorithmic differences** and between a MATLAB®-derived C model and an RTL design?



Case splitting

Decomposition in isolated cases helps in **understanding the root of a bug**



Relaxation of overconstraints

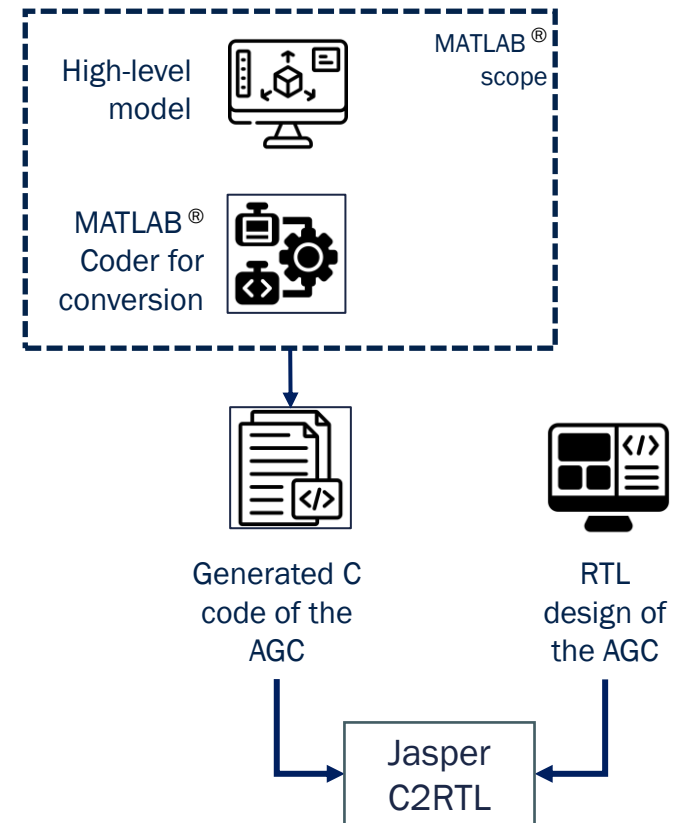
Gradual elimination of overconstraints allows to properly **explore the state space**



Data type check

Datatypes in the generated C code must be **consistent** with the design

Case study: automatic gain control



Results – overview on the verification time

Formal equivalence verification on C-vs-RTL has boosted the verification time

From months to a couple of weeks with respect to UVM dynamic simulations

x9

Environment setup

- Port mapping
- Definition of assumptions

x7

Test development

- Autogenerated assertions
- Manually inserted properties

x2

Debug

- CEX short waves
- Root cause tracking
- Overconstraints relaxation

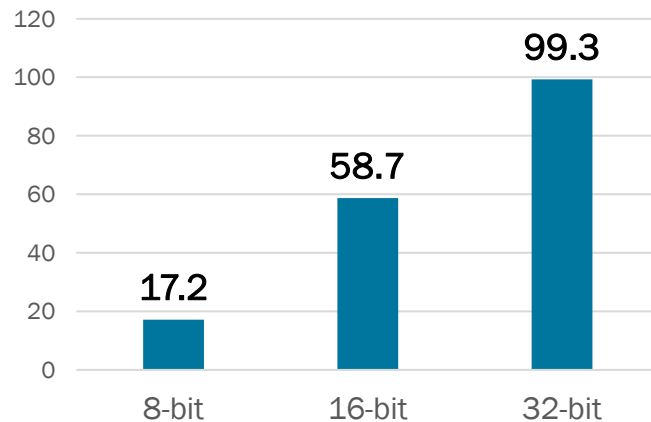
Results – insights on the runtime

It corresponds to the time from the **beginning of verification routines** to the **last proven assertion**

Results obtained with **proof orchestration engine setup**

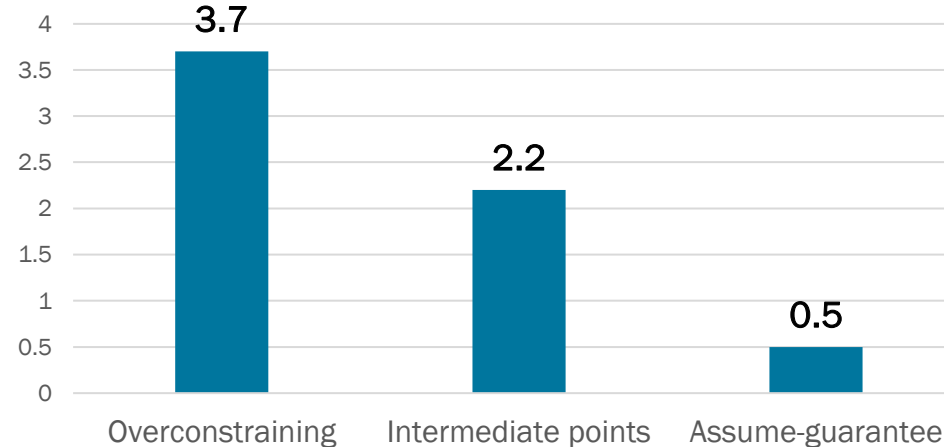
Fixed-Point MAC

Run time (s) vs bit width



Tone generator

Run time (hours) vs formal technique



AGC

	Modes	Time
Complexity ↓	MODE0	0.34 s
	MODE1	9.53 s
	MODE2	121.74 s
	MODE3	754.10 s
	MODE4	1041.33 s
	MODE5*	≈ 48 h
	MODE6*	≈ 48 h

***Bounded proof:** no CEX found, even if the state space is not 100% explored

Conclusions

Drastic **time reduction**
to build and maintain the
verification environment



Short CEX waves make
debug shorter, especially
for subtle corner cases



Exhaustive exploration of the
state space (two bugs were
found in AGC, despite UVM)



**No replacement of our
insight** in DUV expected
behavior and decomposition of
complex properties





JUNE 23-27, 2024

MOSCONE WEST CENTER
SAN FRANCISCO, CA, USA

Thank you!

