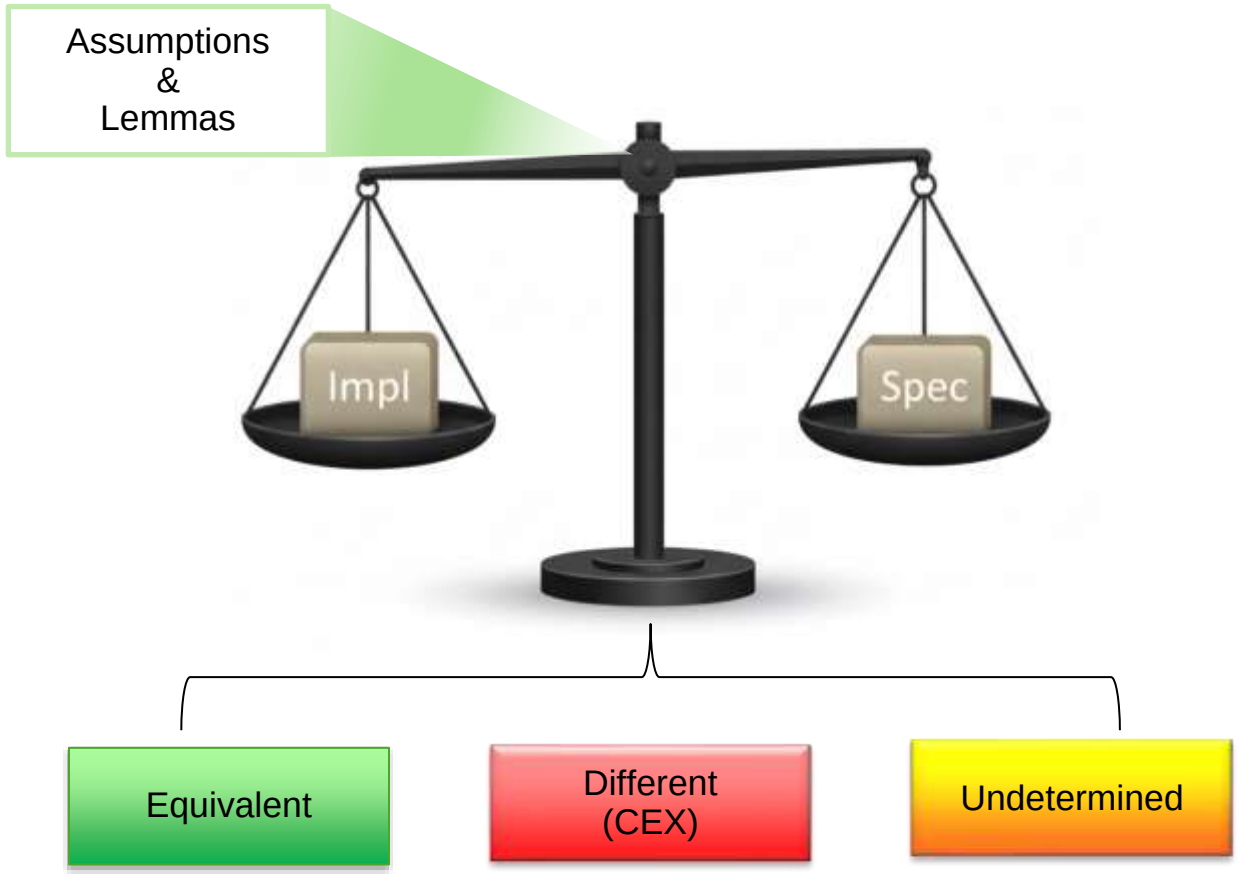
2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

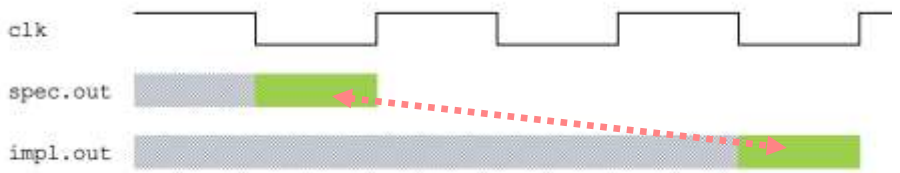
A New Methodology for Formal Equivalence Checking of Sorting Algorithms

Emiliano Morini, Principal FV Engineer, NVIDIA

Formal Equivalence Checking (FEC)

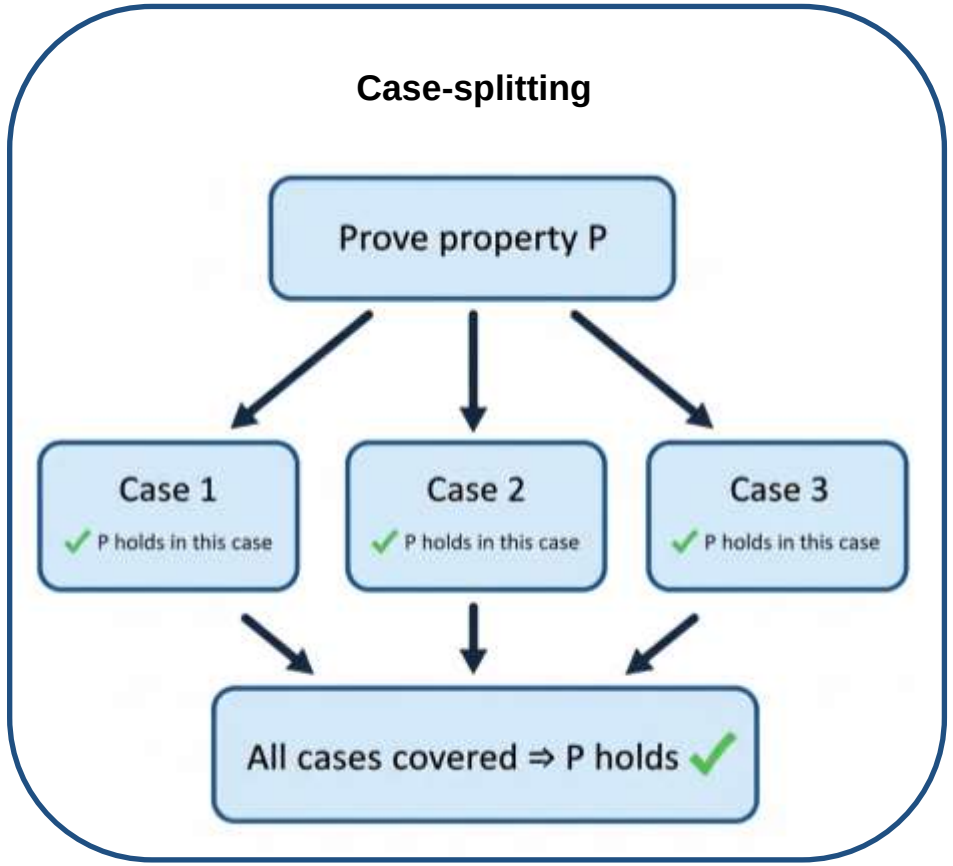


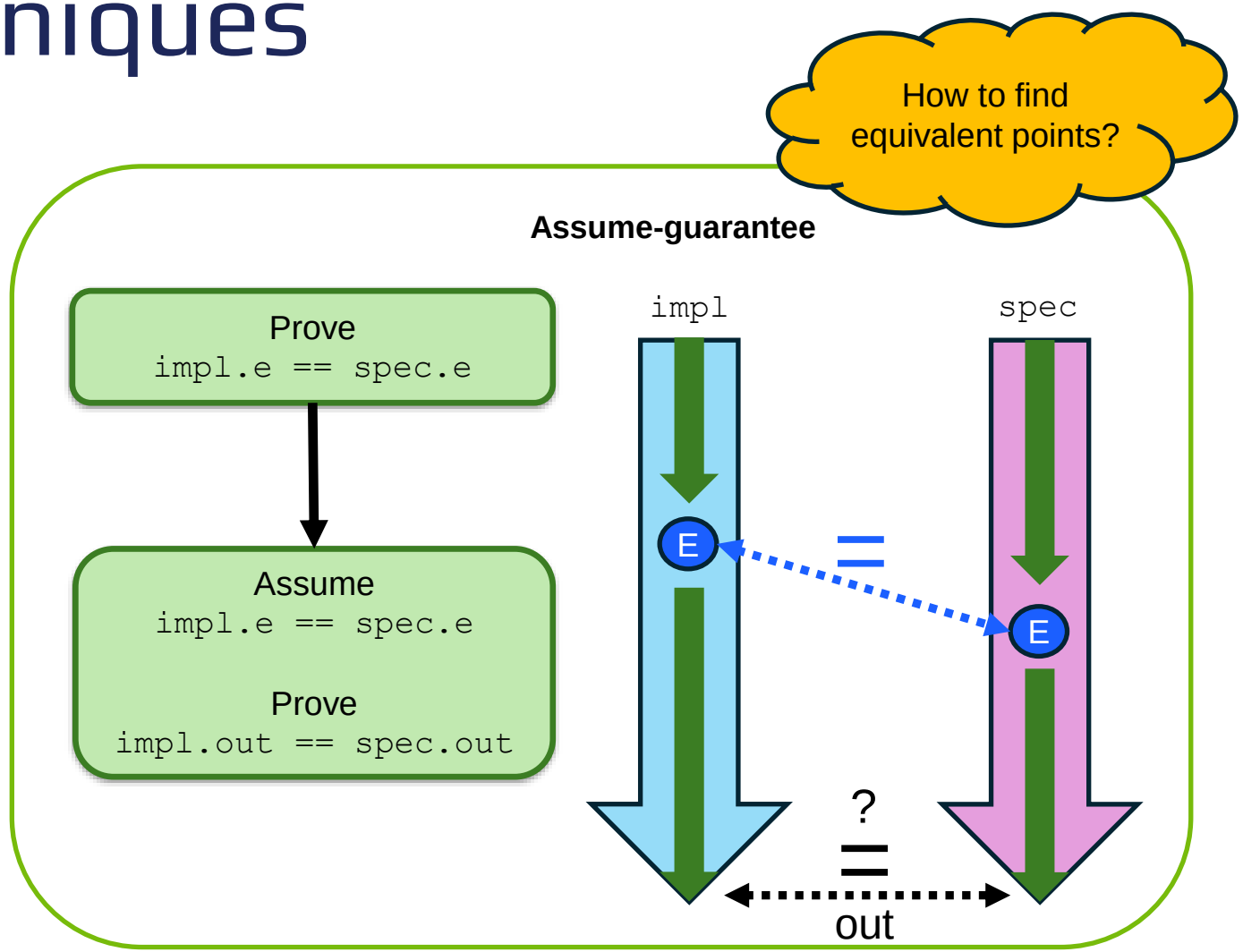
Transactional
Equivalence Checking



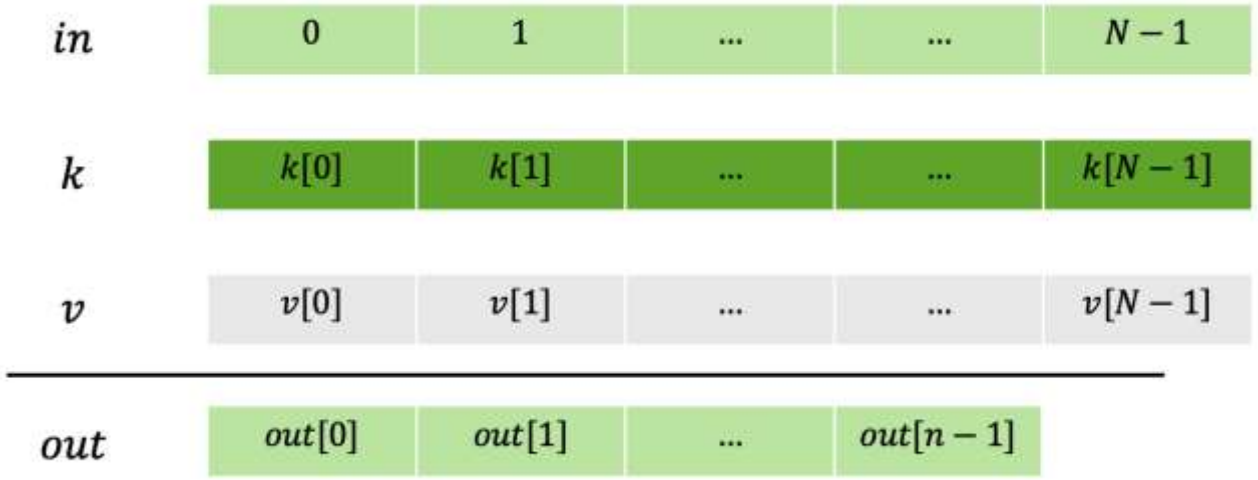
Transactions of varying lengths in implementation and specification

Convergence Techniques





Generalized Sorting Algorithm



N indices to be sorted

Keys for sorting the indices

Validity mask to specify which indices should be considered

First $n \leq N$ sorted valid indices

If $\sum v[i] = m < n,\ out[m], \ldots, out[n-1]$ are undefined

Example
(ascending)

in	0	1	2	3	4	5	$N = 6$
k	2	1	5	0	3	1	$n = 4$
v	0	1	1	0	0	1	
out	1	5	2	*			

Different sorting algorithms produce different intermediate results → Internal equivalent points may not exist

Sorting Abstraction

The specification of a sorting algorithm can be defined using the following properties:

- **RANGE**: Each valid output index is a number between *0* and *N-1*
- **UNIQUENESS**: All valid output indices are distinct
- **VALIDITY**: If m valid input exists $(0 < m \leq n)$, outputs 0 to $m - 1$ are valid
- **ORDERING**: Valid outputs are sorted by keys in ascending order
- **STABILITY**: For equal keys, lower original indices appear first

These properties **univocally** define the output of the sorter

Property Encodings

- **Range**, **Uniqueness** and **Validity**: the encoding is straightforward
- **Ordering** and **Stability**: for each output, we enumerate each index $0 \leq i < N$, and we use a double level of indexing via the output variable

Ordering: $\forall\ 0 \leq i < N$

$$\left(v[out[0]] \wedge v[i]\right) \Rightarrow k[out[0]] \leq k[i]$$

$$\left(v[out[1]] \wedge v[i] \wedge (i \neq out[0])\right) \Rightarrow k[out[1]] \leq k[i]$$

Stability: $\forall\ 0 \leq i < N$

$$\left(v[out[0]] \wedge v[i] \wedge (i \neq out[0]) \wedge \left(k[out[0]] = k[i]\right)\right) \Rightarrow i > out[0]$$

$$\left(v[out[1]] \wedge v[i] \wedge (i \neq out[0]) \wedge (i \neq out[1]) \wedge \left(k[out[1]] = k[i]\right)\right) \Rightarrow i > out[1]$$

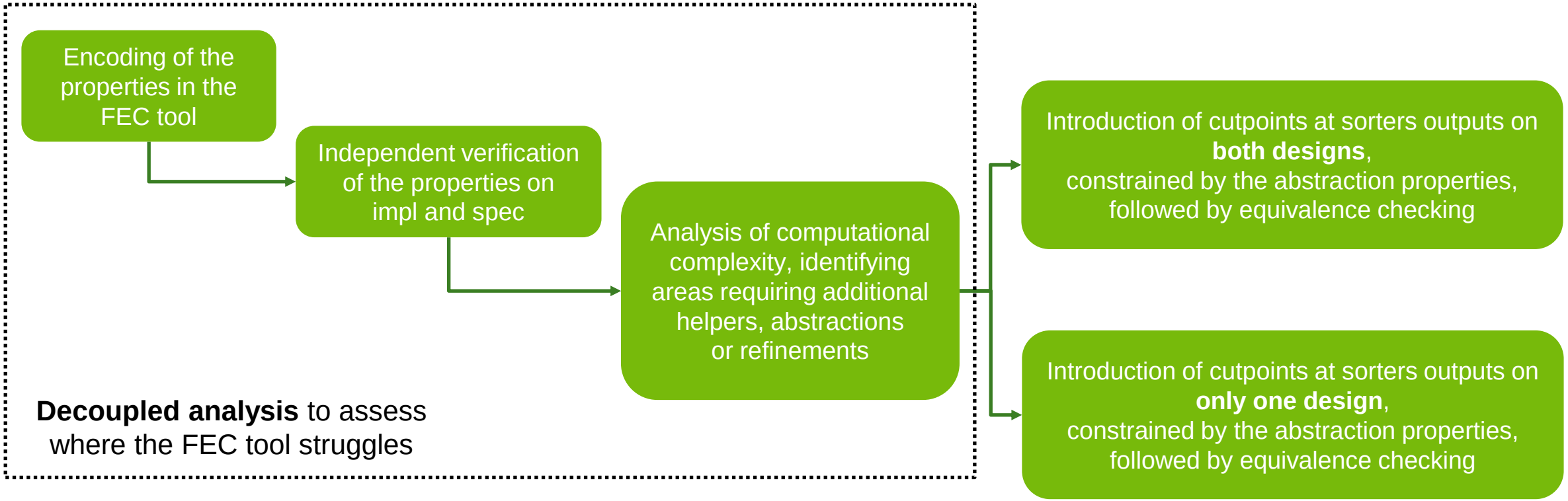
Example for $n = 2$

in	0	1	2	3	4	5	$N = 6$
k	2	1	5	0	3	1	$n = 2$
v	0	1	1	0	0	1	
out	1	5					

Many other encodings are possible

Enumeration helps to identify possible bottlenecks in the verification

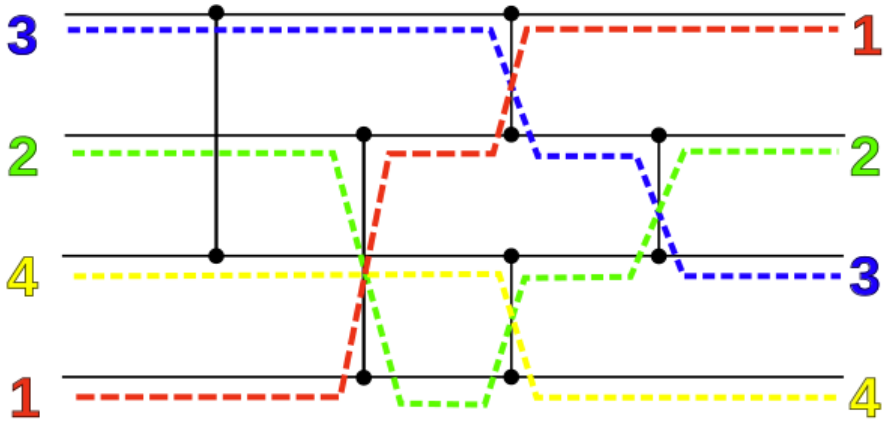
New Verification Methodology



Every step is executed in the same FEC tool, no need to convert properties from another tool

Case Study

RTL: Sorting network



$N = 12$
$n = 4$

C++: Double nested loop

- Find the min among the valid input elements and save it in the output vector
- Mark the min as invalid in the input vector
- Repeat

No internal equivalence points

These sorters are part of bigger systems with different interfaces, so we must verify they receive equivalent input values to apply the new methodology

Previously verified with a large case-splitting on the validity mask, converging in 12+ hours

Case Study - Decoupled Analysis

Upon independently executing the properties for RTL and C++, we observed notable differences in convergence runtime for **Validity** and **Ordering**:

Property type	Convergence Time RTL	Convergence Time C++
Validity	3 sec	20 – 80 min
Ordering	0 – 20 min	33 – 100 min

Time for the 1^{st} output

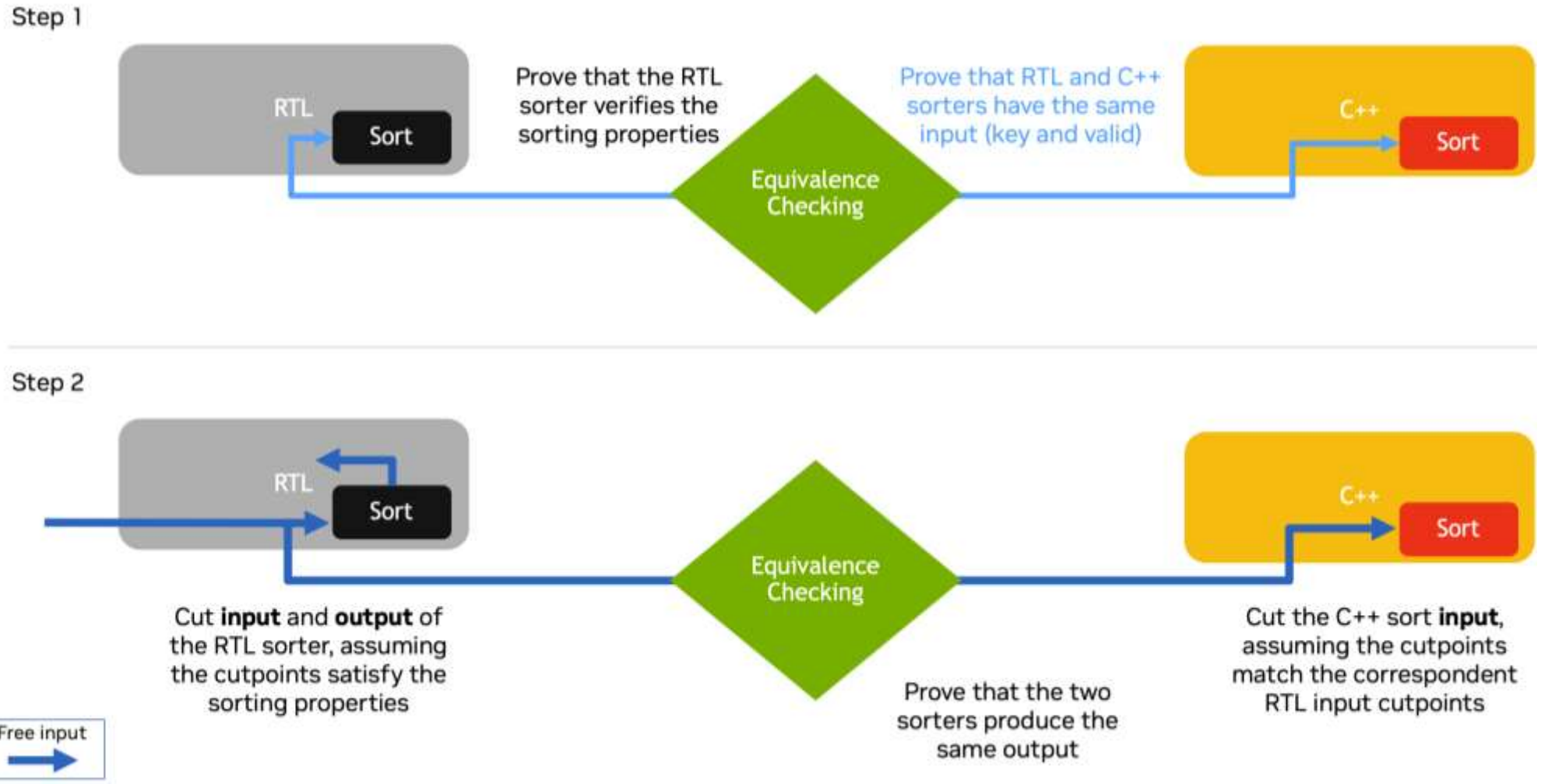
Time for the 4^{th} output

Linear scaling between the different output values

The numbers indicate that the C++ sorter presents greater complexity for the FEC tool

This provides a hint to implement the methodology with cuts only on the RTL output, since it does not require verifying the properties on the C++ side

Case Study - New Proof Strategy



Case Study - Results

Methodology	Runtime (min)	Workers	Machine Memory	Number of subproblems
Case-splitting (old)	760	64	32GB	>1600
Sorting abstraction (new)	27	2	4GB	2

Runtime & Resources **96%** reduction

Additional sanity checks

- ☑ Detection of bugs introduced with mutation testing
- ☑ Failures detected removing any properties

Conclusion

- Directly comparing different sorting algorithms is a challenging task
  - Identifying internal equivalence points to decompose the problem is generally unlikely
  - While a wide case-splitting might eventually succeed, it is resource and time intensive, not scaling well to larger problems
- The newly introduced methodology solves the problem by abstracting the sorting process through a set of properties that uniquely characterize the input-output relationship
  - Utilizing these properties independently for both C++ and RTL enables the identification of specific areas where the verification tool faces difficulties
  - Integrating this abstraction with cutpoints allows the creation of new efficient proofs
  - This approach is tool-independent and compatible with any commercially available FEC tool
- The effectiveness of the approach is demonstrated with a production-level case study that verifies the equivalence of two distinct sorting algorithms and reduces runtime and license usage by 96% compared to the prior proof
- The methodology scales linearly with the number of output values, making it suitable for larger sorting implementations