A New Methodology for Formal Equivalence Checking of Sorting Algorithms

Emiliano Morini
NVIDIA, Santa Clara, USA

***Abstract* - Formal Equivalence Checking (FEC) is an important technique for the verification of datapath components in the semiconductor industry. Although FEC is effectively applied in verifying numerous modules, designs without internal equivalence points, such as sorting algorithms, can pose substantial verification challenges, and proof techniques often do not scale efficiently for larger implementations. This paper presents a novel methodology for FEC of sorting designs, addressing the inherent challenges posed by the algorithmic diversity of possible implementations. The proposed approach introduces a comprehensive set of abstraction properties that uniquely characterize the output of a sorting algorithm, enabling efficient verification without requiring internal equivalence points between implementations. The efficacy of this methodology is demonstrated through a practical case study, where one of our designs was previously verified through an extensive case-splitting approach. The proposed methodology achieves a 96% reduction in runtime while utilizing significantly fewer computational resources and licenses to achieve complete convergence.**

I. Introduction

One of the most important verification methodologies for datapath components in semiconductor design is Formal Equivalence Checking (FEC). This approach primarily relies on commercial Electronic Design Automation (EDA) tools that facilitate the integration of both Hardware Description Language (HDL) and C++ implementations [1], [2], [3]. In scenarios where direct tool convergence proves unattainable, verification engineers often employ an assume-guarantee methodology, wherein internal equivalence points are identified and utilized to decompose the verification problem. This process involves first establishing the equivalence of identified internal signals, subsequently leveraging these relationships as auxiliary assumptions for the comprehensive verification of the design. The importance of internal equivalent pairs is demonstrated by tools having specific procedures to find them, as noted in [6].

While identifying internal equivalence points typically enhances verification efficiency, this approach becomes infeasible when such intermediate equivalence points do not exist. A prime example of this challenge occurs with sorting algorithms, where different implementations can produce identical outputs through fundamentally distinct computational approaches. For such algorithms, establishing internal equivalence points between implementations is inherently impossible. This paper presents a novel methodology for verifying sorting algorithm equivalence through the formal definition of a comprehensive set of properties that abstract the sorter's specification, thereby enabling a decoupled analysis. The main contributions of this work are:

- The definition of sorting properties that uniquely define the sorter behavior and can be tested in a formal tool.
- The usage of such properties to abstract the sorters and prove the equivalence between implementation (RTL) and specification (C++) within a commercial FEC tool.

The advantage of completing both steps in a single tool should not be underestimated; even if the first step can be done in a different tool, e.g. a theorem prover, porting these results into a FEC tool requires a translation step, which is error prone and could affect the soundness of the overall proof.

After a brief introduction to FEC in Section II, the complexities associated with the verification of sorting algorithms are detailed in Section III and the novel methodology for verifying them is presented in Section IV. The efficacy of this approach is demonstrated through a practical case study in Section V, before concluding the paper in Section VI.

II. Formal Equivalence Checking

FEC uses formal methods to establish mathematical equivalence between two distinct design implementations. Given the complexity of modern designs, exhaustive simulation-based verification is computationally infeasible, making formal verification the only viable approach for a complete verification. FEC encompasses three primary categories of equivalence checking:

- *Logic Equivalence Checking*: Assumes a bijective mapping between registers across implementations, subsequently verifying the equivalence of their combinational fan-in logic. This methodology primarily serves to validate the equivalence between Register-Transfer Level (RTL) designs and their synthesized netlists.
- *Sequential Equivalence Checking*: Verifies equivalence between distinct RTL models under identical input sequences and initial states, comparing output values at each clock cycle. This approach is particularly effective for validating local changes, as in the case of clock-gating optimizations.
- *Transactional Equivalence Checking*: Validates functional equivalence between implementations with different latencies and/or languages (typically RTL and C++), focusing on output equivalence at transaction boundaries, where a transaction is defined as the complete computation from input availability to output generation. This methodology is predominantly employed in arithmetic component verification, where usually C++ serves as the specification (golden model) and RTL as the implementation.

This paper focuses on Transactional Equivalence Checking, often referred to as C-to-RTL verification, but it is not limited to it.

Writing C++ and RTL code independently, starting from a common requirements document, significantly reduces the probability of correlated bugs, making C-to-RTL equivalence a successful verification approach for project sign-off. C-to-RTL equivalence verification is also important for performance analysis, as C++ models are frequently employed for pre-synthesis metrics collection and experimental evaluation. The reliability of such analysis depends on the equivalence between the C++ specification and RTL implementation. C++ models offer advantages in verification flexibility, enabling extensive simulation due to superior execution speed compared to RTL. In some cases, these C++ models are verified via theorem proving [8]. In other cases, theorem provers are used to verify directly the RTL code [9].

The FEC methodology has proven effective in verifying fundamental arithmetic operations, including addition, multiplication, fused multiply-add (FMA) [10], division [5], and dot-product [7]. While bug detection typically requires minimal human intervention, establishing the absence of bugs through FEC presents greater challenges. This necessitates sophisticated verification techniques, including case-splitting, assume-guarantee reasoning, and abstraction via cutpoints and black-boxes. For instance, in an FMA verification ($a \cdot b + c$), a common approach involves identifying internal signals representing the extended-precision product $a \cdot b$ in both implementations. The verification process establishes either direct signal equivalence or equivalence relationships under specific conditions (e.g., considering the addition of signals in carry-save format). Subsequently, these signals may be abstracted with cutpoints, and case-splitting may be introduced to handle different cases such as normal numbers, denormals, infinities, and NaNs.

III. Verification of Sorting Algorithms

Sorting operations constitute a fundamental computational primitive across diverse applications in digital design. The verification of equivalence between RTL sorting implementations and their C++ counterparts presents significant challenges due to the possible inherent algorithmic diversity in computing the same output. The absence of equivalent internal points between the two implementations makes traditional equivalence decomposition methodologies less effective. Considering the distinct objectives of RTL designers (power, performance, and area) and C++ developers (speed and reusability), coupled with the need to maintain independence between implementation and specification, it is very likely that the two sides of the comparison use different sorting algorithms to compute the result.

While direct formal verification may prove tractable for small-scale sorting problems, practical applications typically necessitate sophisticated verification strategies.

The formal verification of sorting algorithms has been the subject of prior research. For instance, in [11], the authors employ a deductive theorem prover to verify a sorting algorithm implemented in Java. Their methodology utilizes preconditions and postconditions, aligned with the principles of the Hoare Triple. The authors establish high-level specifications for verification, emphasizing properties such as termination, memory and exception safety, absence of overflow, and ordering correctness. Notably, they do not address equivalence checking and concentrate on sorters that return permutations of the input elements directly, as opposed to valid index arrays, which is the case examined in the subsequent section. Consequently, their verified properties differ from those considered herein.

A. Sorting Abstraction

Consider a C-to-RTL equivalence verification problem involving two distinct sorting implementations. The input consists of N elements, indexed from 0 to $N-1$, where each element i possesses a sorting key value $k[i]$. A binary validity mask $\{v[0], \ldots, v[N-1]\}$ specifies which elements must be considered in the sorting operation. The designs

prioritize valid elements over invalid ones and sort the valid elements according to their keys, returning the first n output indices $\{out[0], \ldots, out[n-1]\}$, where $n \leq N$. For valid elements with the same key, the one with lower index has priority. When $\sum v[i] = m < n$, values $out[m], \ldots, out[n-1]$ are undefined. Without loss of generality, ascending order is assumed; the methodology can be trivially adapted for descending order by negating the sorting keys. This formulation of the problem is not limited by the validity mask and the value n, which accommodate both partial and total sorting scenarios.

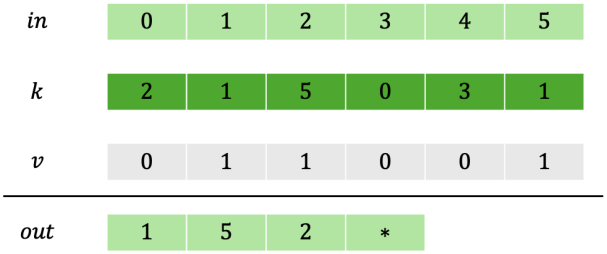

Figure 1. Example of sorter for $N = 6$ and $n = 4$. As there are only $m = 3$ valid elements for this input, the last output $out[3]$ is undefined.

The properties introduced to abstract the output of such a generic sorter are described in Table I.

TABLE I
ABSTRACTION PROPERTIES DEFINING THE OUTPUT OF THE SORTER

Range	Each valid output index is between 0 and $N-1$.
Uniqueness	All valid output indices are distinct.
Validity	If m valid inputs exist ($0 < m \leq n$), 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.

Although not necessary in all the sorting applications, the stability property ensures deterministic output ordering when multiple valid elements possess identical keys. While stable sorting is considered in this paper, our methodology can be extended to handle unstable sorting, properly phrasing the expected relation between the outputs of the two implementations. The abstraction properties account for the case where $\sum v[i] < n$.

Any generic sorter of the type introduced above satisfies the properties listed in Table I: it sorts valid elements, prioritizing the ones with lower index for identical keys, so it satisfies ordering and stability. As the valid indices returned are from the input vector, they are values between 0 and $N-1$, and they are obviously unique, so also range and uniqueness are satisfied. As per specification, it returns n elements, that are all valid if $m \geq n$, or only the first m are valid otherwise. So, for $m \leq n$, the values $out[0], \ldots, out[m-1]$ are valid, meeting the requirement for the validity property.

Vice versa, if a design satisfied the properties of Table I, it prioritizes the valid elements, due to the validity property, sorting them according to their keys, as per ordering definition, prioritizing the ones with lower index when the key is the same, as per the stability property. All the valid input returned a unique and between 0 and $N-1$, matching exactly the specification above.

IV. VERIFICATION METHODOLOGY

The abstraction properties introduced in the previous section enable verification through property-based model checking within the FEC tool. This methodology facilitates the decoupling of verification between implementation and specification, providing valuable insights into the relative complexity of each side of the comparison. By independently verifying these properties on both designs, it is possible to identify which algorithm presents greater challenges for the formal tool, as evidenced by variations in verification runtime and resource utilization.

Consider an equivalence checking problem between two distinct sorting algorithms where direct verification proves inconclusive. Our proposed methodology comprises the following steps:

1. Formal encoding of the abstraction properties in Table I within the FEC framework.
2. Independent verification of the properties on both implementations.

3. Analysis of computational complexity across implementations, identifying where additional helpers, abstractions or refinements are needed, for example using domain reduction techniques, as described in [12].
4. Introduction of cutpoints at sorters outputs on both designs, constrained by the abstraction properties, followed by equivalence verification.

The soundness of this approach relies on the completeness of the abstraction properties. If multiple distinct outputs satisfied these properties, the introduction of cutpoints would enable the formal tool to identify a counterexample by assigning different values to corresponding outputs of the compared designs. In this case, also any deterministic design would fail to prove equivalent to itself.

The methodology can be further refined through selective abstraction, where cutpoints are introduced only on one side output, for example on the RTL design. In this variant, the sorter logic in the RTL is abstracted away and modeled with the properties introduced in Table I. This abstracted RTL implementation is then checked against the unmodified C++ sorter. The choice between one sided or double-sided abstraction depends on the specific characteristics of the implementations. Empirical evaluation of both approaches enables optimization of the verification strategy.

The effectiveness of this methodology relies on the reduction of the verification problem to a constraint satisfaction problem, specifically through the application of abstraction properties. This formulation bears conceptual similarity to well-studied constraint satisfaction puzzles, such as Sudoku and the Non-attacking queens problem, which are frequently employed as introductory examples to formal verification. This approach is not limited to C-to-RTL verification and can be extended also to cases of C-to-C and RTL-to-RTL equivalence checking. Furthermore, this approach does not rely on any tool-specific features, allowing for implementation with any commercially available FEC tools.

B. Properties Encoding

The abstraction properties of Table I can be formally encoded in a FEC tool in different ways. The approach presented here employs an enumeration of properties for each entry position $0 \leq i < N$. For instance, using the notation introduced in Section III-A, the ordering property for a sorter producing $n = 2$ outputs can be expressed through the following logical constraints, $\forall i \in [0, N-1]$:

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

The distinction between equations above is in the additional precondition term in the latter. This supplementary condition is required because i ranges across all possible elements, requiring explicit exclusion of the case where i corresponds to the first output position, which possesses the minimal key value. Similar terms are required for larger values of n.

The stability properties can be similarly formalized; for instance, for output in position one, the property can be expressed as follows, $\forall i \in [0, N-1]$:

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

The encoding of the other properties is similar and is omitted for brevity. While alternative encoding options exist, such as employing symbolic integer variables appropriately constrained, these approaches, though more mathematically elegant, can introduce additional complexity in the verification of the properties, as a symbolic variable would represent multiple values between 1 and N. Such complexity may affect the identification of verification bottlenecks and make the debugging process more challenging, particularly when utilizing commercial tools that do not provide comprehensive internal details to facilitate analysis of property convergence issues.

V. Case Study

To illustrate the effectiveness of our proposed methodology, it is applied to the verification of an industrial RTL design that implements a performance-optimized sorting network. This type of algorithms has been widely used to solve similar problems for many years, with one of the most well-known papers discussing them being [4]. The component under consideration computes the four smallest valid indices from an array of twelve entries. The verification task entails establishing formal equivalence against its C++ reference model. The C++ implementation

employs a double-nested loop algorithm that iteratively identifies the minimum valid element, subsequently marking it as invalid before proceeding to the next iteration.

This case study exemplifies the challenges described in previous sections, as the two implementations lack internal equivalence points that could be utilized for problem decomposition. Furthermore, the sorters are embedded within larger designs, exhibiting not exactly matching interfaces and requiring preprocessing functions for input data preparation. While these factors introduce additional complexity, they do not preclude the application of our methodology.

In previous project iterations, this verification constituted one of the most computationally intensive tests in our regression suite, requiring significant runtime, licenses, and computational resources. As anticipated, the verification tools failed to achieve convergence without additional guidance. Given the absence of internal equivalence points, the initial verification approach employed an extensive case-splitting on the validity mask values. For each specific validity mask configuration, the tool successfully established equivalence between the two sides of the comparison. While the tool could handle multiple cases simultaneously for configurations with few valid inputs, more complex scenarios necessitated individual case analysis. An alternative faster proof using data domain reductions was implemented, but it required additional manual checks to confirm its applicability, so it was not pursued further. Time constraints imposed by release scheduling precluded the development of alternative methodologies at that time.

Although the final proof achieved convergence, concerns regarding scalability persisted, particularly for potential future modifications involving larger input arrays or increased number of output values. These considerations motivated the development of this new approach.

Following the definition of the properties in Table I, a decoupled verification analysis to assess the relative complexity of each implementation was conducted. The results revealed that the C++ implementation presented greater verification challenges, as evidenced by significant runtime differences. The most interesting runtime differences are shown in Table II. The time interval refers to the time required by the first and the last property of a specific type to converge. This experiment also showed a linear scaling of the properties' convergence time with respect to the number of output values.

TABLE II
DECOUPLED ANALYSIS RESULTS 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

Based on these findings, a verification approach comprising the following steps was implemented:

1.a) Verification of abstract sorting properties of Table I on the RTL sorter.

1.b) Establishment of input equivalence between RTL and C++ sorters.

2.a) Selective introduction of cutpoints at the RTL sorter inputs and outputs, constrained by assuming the sorting properties.

2.b) Equivalence checking after the application of cutpoints to the C++ sorter input, assuming the equality with the corresponding RTL sorter input cutpoints.

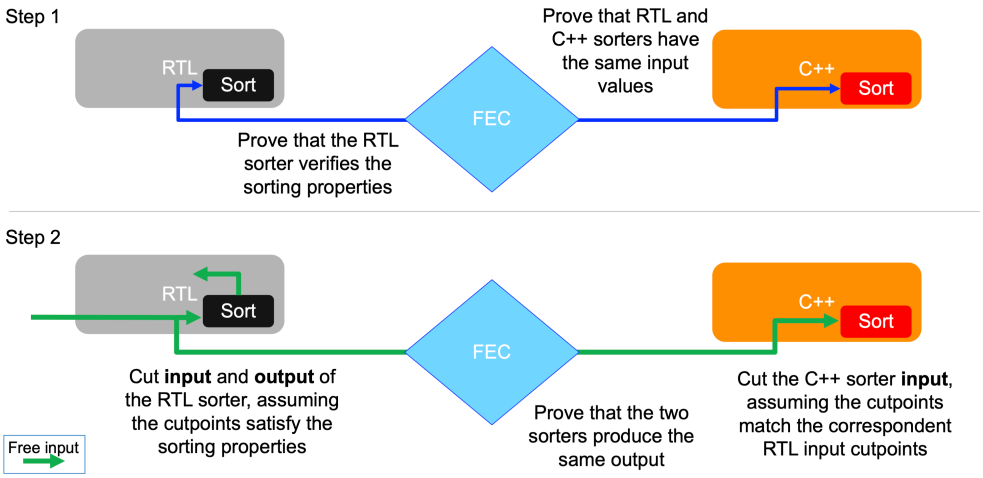

Figure 2. Two-steps verification proof for the case study.

This methodology, illustrated in Figure 2, is implemented through two distinct proofs. The first proof establishes the sorting properties and input equivalence of the sorters, ensuring the soundness of the cutpoint assumptions in the subsequent proof. The second proof introduces cutpoints constrained by the properties proven in the first proof, enabling the FEC tool to prove that the C++ implementation produces the same unique output satisfying the abstraction properties proven for the RTL sorter. This two-steps approach establishes complete functional equivalence between RTL and C++.

A quantitative comparison between the original and proposed methodologies is presented in Table III. Both verification approaches were executed in a distributed computing environment with configurable memory allocation and workers count for parallel proof execution. The original methodology required 760 minutes for convergence, utilizing 64 workers on machines with 32GB memory each. This approach employed three sub-proofs, with the final proof implementing a large case-splitting on the validity mask, generating over 1600 subproblems in total. In contrast, the new methodology achieved convergence in under 30 minutes using two workers on 4GB memory machines, generating only the two proofs described in Figure 2. These results demonstrate a 96% reduction in runtime while requiring only 4% of the computational resources. The improvements allow rapid re-verification following design modifications, while the linear scaling of properties convergence time enables the methodology's applicability to larger sorting networks.

TABLE III
VERIFICATION METHODOLOGIES COMPARISON

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

To validate the methodology's effectiveness in detecting violations, mutation testing [13] was applied to systematically introduce bugs into the designs. These were promptly identified by the FEC tool, which generated appropriate counterexamples within seconds. The types of bugs inserted included alterations to comparison operators (e.g., interchanging "<" with "≤"), adjustments to the application of validity masks, and commenting out selected lines of code within the core sorting procedure.

Furthermore, it was confirmed that the properties defined in our setup, according to Table I, are not only sufficient to prove the equivalence, but also necessary. The assessment involved iteratively performing Step 2 of our proof

strategy, each time excluding one of the encoded properties. In each instance, the FEC tool identified a counterexample, thereby confirming that the diminished set of properties was insufficient to model the problem.

VI. Conclusions and Future Work

This paper presents a novel methodology for FEC of sorting designs, addressing the inherent challenges posed by the algorithmic diversity of possible implementations. The proposed approach introduces a set of abstraction properties systematically derived to uniquely characterize the output of a sorting algorithm, enabling efficient verification and solving the problem of the absence of internal equivalence points between the two sides of the comparison. This methodology represents a significant advancement over traditional case-splitting approaches, and it was demonstrated through a relevant case study. The conventional methodology, employing an extensive case-splitting on the validity mask, required approximately 12.5 hours to achieve convergence, using 64 workers. In contrast, the new abstraction-based approach achieves verification in under 30 minutes using only two workers, representing a 96% reduction in both runtime and resource requirements. This approach not only reduces computational complexity but also provides valuable insights into the relative verification challenges of different algorithmic implementations. Furthermore, the methodology exhibits linear scaling with respect to the number of output values, suggesting its applicability to larger sorting implementations.

The results presented in this paper demonstrate that FEC of sorting algorithms can be significantly improved through careful abstraction and property-based verification within the FEC framework. Future work will extend the benchmarking, comparing multiple FEC tools and verifying different parameterizations of the sorting algorithms. The application of similar abstractions to other classes of algorithms that lack internal equivalence points will also be explored, potentially leading to broader improvements in formal verification practices. For this purpose, an AI-based approach could be investigated to produce abstraction properties analogous to those presented in Table I.

Acknowledgment

The author expresses sincere gratitude to the members of the NVIDIA GPU Formal Verification Team for their encouragement, support, and valuable suggestions that contributed to the completion of this paper.

References

[1] Jasper C2RTL App. https://www.cadence.com/en US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-c-formal-verification.html. Accessed: 2025-03-19.

[2] SLEC System. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls-verification/slec/. Accessed: 2025-03-19.

[3] VC Formal Datapath Validation. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal/vc-formal-datapath-validation.html. Accessed: 2025-03-19.

[4] K. E. Batcher. Sorting networks and their applications. In Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring), page 307314, New York, NY, USA, 1968. Association for Computing Machinery.

[5] Ashish Kapoor, Warren Ferguson, Himanshu Jain, and Sudipta Kundu. Formal Verification of Floating-Point Division. In 2023 IEEE 30th Symposium on Computer Arithmetic (ARITH), pages 93–96, Los Alamitos, CA, USA, September 2023. IEEE Computer Society.

[6] Alfred Koelbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to rtl equivalence checking. In 2009 Design, Automation & Test in Europe Conference & Exhibition, 2009.

[7] Emiliano Morini, Bill Zorn, Disha Puri, Madhurima Eranki, and Shravya Jampana. Achieving end-to-end formal verification of large floating-point dot product accumulate systolic units. In 2024 Design, Automation &Test in Europe Conference & Exhibition, 2024.

[8] David Russinoff, Javier Bruguera, Cuong Chau, Mayank Manjrekar, Nicholas Pfister, and Harsha Valsaraju. Formal verification of a chained multiply-add design: Combining theorem proving and equivalence checking. In 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH), pages 120–126, 2022.

[9] Mertcan Temel, Anna Slobodova, and Warren A. Hunt. Automated and scalable verification of integer multipliers. In Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, 1CA, USA, July 2124, 2020, Proceedings, Part I, page 485507, Berlin, Heidelberg, Springer-Verlag, 2020.

[10] Bin Xue, Prosenjit Chatterjee, and Sandeep K. Shukla. Simplification of c-rtl equivalent checking for fused multiply add unit using intermediate models. In 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC), pages 723–728, 2013.

[11] Bernhard Beckert, Peter Sanders, Mattias Ulbrich, Sascha Witt and Julian Wiesler. Formally Verifying an Efficient Sorter. In: Finkbeiner, B., Kovács, L. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2024. Lecture Notes in Computer Science, vol 14570. Springer, Cham, 2024.

[12] Robert McKemey, Sam Elliott, Emiliano Morini, Max Freiburghaus. Verifying a hardware design for a component that implements a permutation respecting function, US PATENT 11455451, 2022/9/27.

[13] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, Mark Harman. Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Elsevier, Volume 112, pages 275-378, 2019.