A Directed Test Generator for Shared-Memory Verification of Multicore Chip Designs

Gabriel A. G. Andrade, Marleson Graf, Nícolas Pfeifer, and Luiz C. V. dos Santos

***Abstract*—The functional verification of multicore chips requires the generation of parallel test programs able to expose design errors and ensure high coverage in less time. Albeit the coherence hardware can scale gracefully as the number of cores grows, the state space of the coherence protocol increases exponentially. That is why this article describes a directed test generation approach that exploits random test generation (RTG) for avoiding explicit enumeration of the coherence state space while memory consistency is verified. The novel approach was designed for synergy between a data-driven engine that explores neighborhoods toward higher coverage and a model-based engine that exploits constraints while driving RTG toward faster coverage evolution. As compared to a state-of-the-art data-driven generator and to a model-based generator, the proposed approach led to superior coverage evolution with time, when targeting 32-core designs relying on different protocols. For MOESI 2-level, the novel approach was from 4.8 to 18.7 faster to reach the data-driven generator's maximal coverage, and it was up to 2.7 faster to reach the model-driven generator's. For MESI 3-level, it found, in 10 to 15 min, a few errors whose detection required the data-driven generator 45 min to 7 h.**

***Index Terms*—Coherence, design aids, shared memory, single-chip multiprocessors, test generation, verification.**

I. Introduction

THREE architectural trends challenge the generation of parallel programs for multicore design verification: *on-chip cache coherence* can scale gracefully as the number of cores grows [1], and coherent shared memory is expected even in the scale of hundreds of cores [2]; *multicore scaling*, however, exponentially increases the state space of the coherence protocol, and *relaxed consistency* is adopted by most manufacturers. Thus, verification techniques should not enumerate that state space, should exploit coherence properties to reduce the complexity, and should handle the consistency requirements of sophisticated architectures (e.g., IBM Power9, ARMv8, and RISC-V).

There are three generation approaches to shared-memory verification: 1) *litmus test generation*; 2) constrained random test generation (RTG) combined with runtime checking; and 3) directed test generation (DTG). The first one exploits a memory consistency model (MCM) [3] to synthesize programs able to check consistency rules by means of execution witnesses. Albeit quite efficient to find design errors [4]–[6], its coverage control is limited [7]. The second one exploits randomness to raise the coverage of memory events and lets an independent checker verify MCM rules on the fly [8], [9]. Since test generation is not tied to the MCM, it can exploit constraints to improve coverage. However, only the third approach provides actual coverage control.

This article proposes an approach to DTG that is based on coverage-driven RTG. It is part of a functional verification framework that was designed to be reusable across derivative designs, protocol variants, and distinct coverage metrics.

Fig. 1 depicts the verification framework. The generator consists of a Directing Engine that commands an RTG engine, which relies on three parameters to constrain the building of a test program: 1) the number of memory operations (n); 2) the number of shared memory locations (s); and 3) the number of distinct cache sets to which the locations can be mapped (k). The framework includes a simulator for a full-system design representation of a multicore chip. While the simulator executes a test program, monitors observe memory events at relevant points of each core domain. A runtime checker analyzes the monitored events according to the axioms of a given MCM. Besides, other monitors observe events that serve as coverage witnesses from which a Coverage Analyzer computes the cumulative coverage of all tests executed so far. The Directing Engine takes coverage into account before selecting the next setting of parameters for the RTG engine.

This article proposes the Directing Engine shown in Fig. 2. It consists of a generation space *Explorer* and a test generation *Driver*. The Explorer assumes bounds on test sizes ($n_{\min}, n_{\max}$) and on the amount of shared locations ($s_{\min}, s_{\max}$). It defines an encoding (G) consisting of multiple settings for the parameters (n, s, k) for inducing a test suite. The Driver serializes them to induce an order of random tests leading to fast coverage evolution. The Explorer defines new encodings as far as coverage is not satisfactory, and returns the one leading to the best solution (G_{opt}).

Manuscript received August 24, 2019; revised December 6, 2019; accepted February 7, 2020. Date of publication February 17, 2020; date of current version November 20, 2020. This work was supported in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico, Brazil (CNPq) under Grant 141686/2019-7, and in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brazil—Finance Code 001. This article was recommended by Associate Editor J. Cortadella. *(Corresponding author: Gabriel A. G. Andrade.)*

Gabriel A. G. Andrade is with the Automation and Systems Engineering Graduate Program, Federal University of Santa Catarina, Florianópolis 88040-900, Brazil.

Marleson Graf and Nícolas Pfeifer are with the Computer Science Graduate Program, Federal University of Santa Catarina, Florianópolis 88040-900, Brazil.

Luiz C. V. dos Santos is with the Computer Science Graduate Program, Federal University of Santa Catarina, Florianópolis 88040-900, Brazil, and also with the Automation and Systems Engineering Graduate Program, Federal University of Santa Catarina, Florianópolis 88040-900, Brazil.

Digital Object Identifier 10.1109/TCAD.2020.2974343

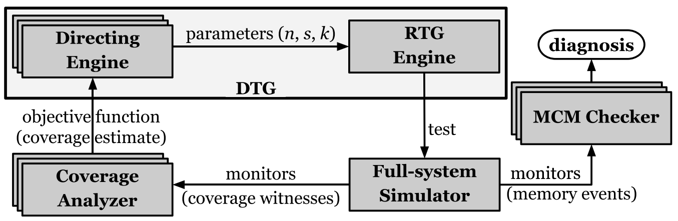


Fig. 1. Overview of the verification framework.

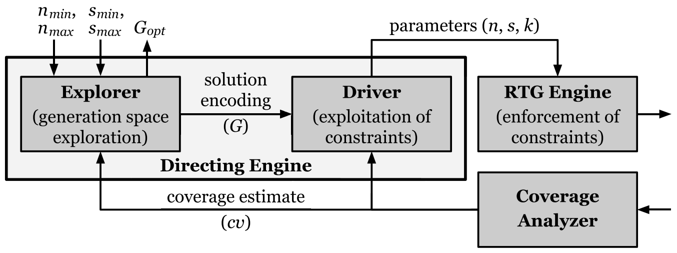


Fig. 2. Anatomy of the proposed Directing Engine.

The main contributions of this article are as follows.

1) The formulation of DTG as a double-objective optimization problem deliberately defined in the RTG space.
2) A novel data-driven technique that explores neighborhoods of the RTG space (instead of enumerating the protocol state space as in [10]–[12]), and avoids excluding optimal solutions from the search space.
3) A generalization of an early model-based technique [13] such that constraints can be exploited for fast coverage evolution without hampering neighborhood exploration.

The first contribution allows a pragmatic way of building upon the legacy of constrained RTG, but under a novel *hybrid approach* (enabled by the other contributions). Instead of a plain composition of well-known techniques, the approach was designed for proper decoupling of competences: *data-driven exploration* and *model-based exploitation*. The synergy between its subengines is demonstrated by experimental results.

II. Related Work

Table I splits DTG approaches into two main classes. *Model-based generators* rely on some abstraction (e.g., FSM or graph) that either explicitly encodes what has to be covered (e.g., the states or transitions of a coherence protocol [10]) or implicitly captures how coverage can be increased [13]. *Data-driven generators* rely on feedback loops to adaptively change generation parameters according to coverage evolution.

An approach focuses on coherence protocol verification [10]. It builds sequences of instructions that induce an Euler tour on a graph representing the state space. Since it relies on the product of FSMs at each core domain, such approach was not scalable. It deserved a recent extension [12], based on a symmetry reduction technique, which defines equivalent classes and restricts the state space to class representatives, allowing a tradeoff between coverage and verification budget. The approach assumes that transitions between stable and transient states are correct. Such assumption is not suitable for memory consistency verification, because it requires artificial order constraints[1] for proper controllability, thereby inhibiting data races, which are well-known mechanisms for exposing shared-memory design errors [8], [15]. Besides, the approach leads to abstract transitions that may aggregate multiple paths over transient states, but only one of them will be covered by the Euler tour, limiting error discovery. Since the approach explicitly encodes *what* has to be covered and fixes the way of traversal, it cannot exploit coverage evolution dynamically.

TABLE I
DTG Approaches for Functional Verification of Multicore Chips

Approach	Scope	Directing Engine
[10] Qin & Mishra (2012)	coherence protocol	Euler tour
[12] Lyu et al. (2018)	coherence protocol	Euler tour, quotient space
[13] Andrade et al. (2018)	shared memory	coverage model
[14] Fine et. al. (2006)	full system	Bayesian network
[11] Wagner & Bertacco (2008)	coherence protocol	multiple intelligent agents
[7] Elver & Nagarajan (2016)	shared memory	genetic programming

Bayesian networks were exploited for DTG [14]. Statistical inference is used to build a Bayesian network for providing the most probable generator settings that would achieve a certain coverage goal. This technique requires a training phase to establish the basis for future decision making. However, the need for training may become a drawback, unless its contribution to the overall effort can be kept negligible.

Mcjammer [11] decomposes the product FSM into dichotomic FSMs that capture the protocol behavior from the perspective of each core domain. Multiple intelligent agents, each working at a distinct core domain, cooperate to improve the overall transition coverage.

Genetic Programming offers another approach for building new tests from old ones. For instance, McVerSi [7] tailors the fitness function for improving memory consistency verification. To obtain a new population from the fittest tests, it employs a selective crossover function that favors the selection of memory operations contributing to higher nondeterminism.

Our prior model-based work [13] focused on ordering random tests to favor coverage evolution, but it did not allow *dynamic* coverage control, and it relied on *greedy* heuristics that induced fixed neighborhoods, which hampered faster evolution when exploring the RTG space. This motivated the more general formulation proposed in the next section and the hybrid approach proposed in Section V, where a data-driven engine focuses on coverage-controlled neighborhood exploration and a generalized model-driven engine focuses on ordering the tests induced by a given neighborhood.

III. DTG Formulation as Optimization Problem

Let N and S be the sets of allowed values for the parameters n and s (respectively) that are within user-defined bounds and are induced by the range of a function[2] $f(i) = 2^i$, and let the values allowed for parameter k be bounded for each allowed

[1]It employs thread barriers to create a global serialization of all instructions.

[2]This could be replaced by another function without loss of generality, as far as the *perturbation* function defined in Section V-A is accordingly adjusted.

value of s and be constrained to be multiples[3] of it, as follows:

$$N = \{n : n_{\min} \le n \le n_{\max} \wedge n = 2^i \text{ for some } i \in \mathbb{N}\}$$
$$S = \{s : s_{\min} \le s \le s_{\max} \wedge s = 2^i \text{ for some } i \in \mathbb{N}\}$$
$$K = \{k : (1 \le k \le s) \wedge (s \in S) \wedge (s \mod k = 0)\}.$$

Each *test* to be synthesized by the RTG engine is specified by a setting of the parameters corresponding to a point in a 3-D *generation space* $\mathbb{G} = N \times S \times K$. A collection of such points induces the generation of a *test suite*.

Problem 1: Given $\mathbb{G} = N \times S \times K$, find an ordered subspace $G_{\text{opt}} \subseteq \mathbb{G}$ that maximizes coverage in minimum time.

Note that Problem 1 ranks coverage first among the two objectives. The most important difference between Problem 1 and conventional coverage-driven RTG instances is the finding of an *ordered* subset of constraints, because that is the key to shaping coverage evolution, as shown in the next section.

IV. Directing Engine at Work: Example

We encode every candidate solution as a subset $G \subseteq \mathbb{G}$. Our Directing Engine iteratively explores solutions induced by a neighborhood function. It combines the current solution and a neighbor into a new solution whose coverage is not inferior.

To illustrate how it works, let us arbitrarily select an encoding (G) representing a solution, as shown in Fig. 3(a). The x-axis displays all pairs in $S \times K$ induced by a range of locations between 4 and 32. The y-axis shows all values in N (in logarithmic scale) for a range of operations between 1Ki and 8Ki. The Explorer provides the encoding G to the Driver, which serializes its points (as depicted by the numbering above the marks) in an order that speeds up coverage evolution. Such serialization is induced by alternately favoring replacement and collision events. A *replacement* results from accesses to distinct locations that happen to induce the eviction of a block from cache. A *collision* [16] results from accesses to the same location. As detailed in Section V-B, an odd (even) number represents a test where replacements (collisions) are favored.

From the current solution (G), the Explorer generates an encoding for a neighbor (Q). Fig. 3(b) illustrates the encoding of one *possible* neighbor of G. A few of the neighbor's points may coincide with the current solution's. Such points are not sent to the Driver, as shown by the unnumbered marks. This avoids re-executing the same tests.

Then the Explorer combines the current solution and its neighbor into a new solution that certainly does not lead to inferior coverage. A possible outcome of such combination is depicted in Fig. 3(c). From this new solution, the entire process is repeated until a termination criterion is satisfied.

V. Directing Engine: Algorithms

Instead of trying to find the optimal subspace G_{opt}, we rely on local search for an approximate solution. We let $V \subset \mathbb{G}$ denote the collection of points that have been *visited*, and we let $cv(X)$ denote the cumulative coverage of all tests induced by an arbitrary collection $X \subset \mathbb{G}$.

[3]This constraint results in uniform competition of locations for cache sets, which benefits coverage control, as will be explained in Section V-B.

Algorithm 1 Directing Engine $(G_0, n_{\min}, n_{\max}, s_{\min}, s_{\max})$

```
1: $\mathbb{G} \leftarrow$ GENERATION SPACE$(n_{\min}, n_{\max}, s_{\min}, s_{\max})$
2: $V \leftarrow \emptyset$
3: if $G_0 = \emptyset$ then
4:     $G_0 \leftarrow \{(n,s,k):(n,s,k) \in \mathbb{G} \wedge n = n_{\min}\}$
5:     $s_{chosen} \leftarrow$ RANDOM$(\{s:(n,s,k) \in G_0\})$
6:     $G_0 \leftarrow \{(n,s,k):(n,s,k) \in G_0 \wedge s \le s_{chosen}\}$
7: $G \leftarrow$ EXPLORER$(G_0)$
8: return $G$
```

Algorithm 2 Explorer(G)

```
1: DRIVER$(G)$
2: repeat
3:     $Q \leftarrow$ GENERATE NEIGHBOR$(G)$
4:     if $Q \setminus V = \emptyset$ then
5:         $Q \leftarrow$ GENERATE NEIGHBOR$(V)$
6:     DRIVER$(Q \setminus V)$
7:     $G \leftarrow$ REDUCE NEIGHBOR$(G, Q)$
8: until $cv(V) = 1 \vee$ TIMEOUT$() \vee (V = \mathbb{G})$
9: return $G$
```

Algorithm 3 Reduce Neighbor(G, Q)

```
1: $G' \leftarrow \emptyset$
2: for all $(n,s,k) \in G \cup Q$ do
3:     if $cv(G' \cup \{(n,s,k)\}) > cv(G')$ then
4:         $G' \leftarrow G' \cup \{(n,s,k)\}$
5: return $G'$
```

Algorithm 4 Generate-Neighbor(G)

```
1: let $D$:$\mathbb{G} \rightarrow$ {TRUE, FALSE} be an empty dictionary
2: $Q \leftarrow \emptyset$
3: for each $(n,s,k) \in G$ do
4:     $D \leftarrow$ SELECT-NEIGHBORING-POINTS$((n,s,k), D)$
5: for each $(n,s,k) \in D$ do
6:     if $D[(n,s,k)] =$ TRUE then
7:         $Q \leftarrow Q \cup \{(n,s,k)\}$
8: return $Q$
```

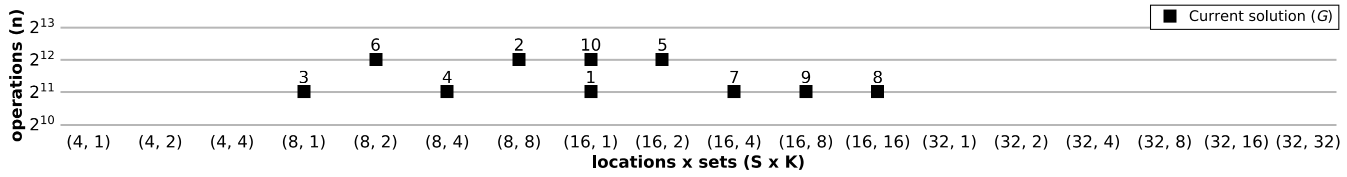
From user-defined bounds, Algorithm 1 induces the generation space (line 1), as described in Section III. If no initial solution is provided, it randomly defines one (lines 4–6) from which local search is launched (line 7).

A. Data-Driven Explorer

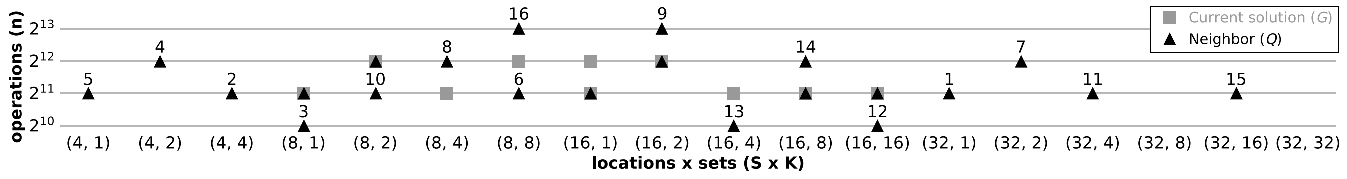
Algorithm 2 iteratively generates neighboring solutions (line 3), and keeps the best solution so far (line 7) until a termination criterion is satisfied (line 8). It invokes the Driver for the initial solution (line 1) and for every generated neighbor (line 6). To save time, it invokes the Driver only to points of the neighbor that have never been visited before. Between such major steps, it treats the cases where the neighbor is either singular or redundant (lines 4 and 5), i.e., either all candidate points happened to be out of bounds or have already been visited.

Algorithm 3 combines the points of the current solution and its neighbor (i.e., $G \cup Q$) and reduces them to a subset (G') that certainly has the same cumulative coverage.

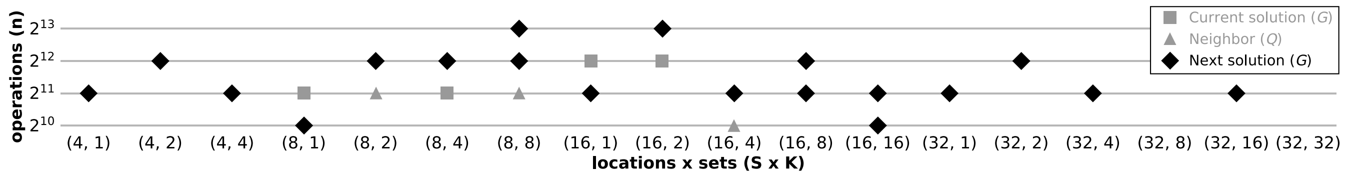
From the current solution (G), Algorithm 4 generates another solution (Q) according to a neighborhood function.



(a)



(b)



(c)

Fig. 3. Explorer and Driver at work: how the exploration of neighbors influences the exploitation of constraints within the Directing Engine. (a) Current solution. (b) Neighbor solution. (c) Next solution.

Algorithm 5 Select-Neighboring-Points($(n, s, k), D$)

```
1: for each $p \in \pi((n, s, k))$ do
2:     if $p \in \mathbb{G} \wedge p \notin D$ then
3:         $D[p] \leftarrow$ RANDOM({TRUE, FALSE})
4: return $D$
```

It first defines an empty dictionary (D) to register which candidate points will be used (or not) for encoding the neighbor (line 1). Then, for each point belonging to the current solution, it selects candidate points for the neighbor (lines 3 and 4). Finally, the neighbor is encoded with the selected points (lines 5–7).

Our approach uses a function to map a point of the current solution to candidate points for the neighbor, as follows.

Definition 1: The *perturbation* function $\pi : \mathbb{G} \to \mathcal{P}(\mathbb{G})$ maps each point (n, s, k) to a subset of the generation space $\{(2n, s, k), (n, 2s, k), (n, s, 2k), (n/2, s, k), (n, s/2, k), (n, s, k/2)\}$.

Note that π can induce whatever point of the generation space, thereby not excluding optimal solutions *a priori*. However, multiple criteria could be used for the *selection* of neighboring points. Without loss of generality, but for compactness, this article shows a single way of tailoring the neighborhood. Given a point (n, s, k) of the current solution, Algorithm 5 randomly selects neighboring points that are within bounds.

B. Model-Based Driver

The Driver dynamically exploits constraints when commanding the RTG engine[4] to induce successive tests. The choice of constraints relies on a model that (pessimistically) assumes that transitions are induced either by replacements or by collisions. The Driver applies a constraint to induce a test favoring replacements and then a different constraint to induce the next test favoring collisions. The alternation between these two types of bias makes transitions *less likely to be revisited*, which favors coverage evolution. To induce such alternation, the Driver lowers the probability of transitions from one type in an attempt to raise the probability of transitions of the other type. The control on replacement events is the key to enabling such alternation, as follows.

For higher controllability, we enforce uniform competition. Remind that k defines the number of distinct cache sets for which the s shared locations compete. For each value of s, we constrain the values that can be assigned to k such that exactly s/k locations compete for each cache set. Such *uniform* distribution maximizes the probability of controlling replacements in all sets for a given setting of s.

Let α denote the associativity of a cache. For inducing a replacement event in a given cache set, a sequence of at least $\alpha + 1$ references to *distinct* locations competing for that set is required. Therefore, a necessary condition for enabling replacement is $s/k \geq \alpha + 1$. Conversely, a sufficient condition for disabling replacement in all sets is $s/k < \alpha + 1 \Leftrightarrow s/k \leq \alpha$. Thus, there is a threshold s/α for the value of k above which replacement is certainly disabled, but below which it may be enabled depending on the sequence of references that turns out to be generated randomly. Note that such threshold is different for hierarchical levels with typically distinct degrees of associativity. This indicates that multiple values of k should be exploited for controlling transitions in the FSMs at all levels.

Therefore, the Driver can stimulate the intended alternation by selecting values of k that enable or disable replacement events. Such selection relies on the following model.

[4]The RTG engine (in its turn) enforces such constraints with a *biasing* RTG technique [17] (which is outside the scope of this DTG-oriented article).

Let $N_R(set(a))$ denote the average number of replacement events for the cache set assigned to the memory block where location a resides. An increase in $N_R(set(a))$ raises the probability of transitions induced by replacements. We have to count the number of replacement events, which depends on the associativity and on the memory access pattern. The ratio n/k measures how many operations are mapped to the same cache set on average. A best-case access pattern for replacement events is such that every element of a sequence of n/k accesses makes reference to distinct locations mapped to the same set. Replacement takes place at every $\alpha+1$ such accesses. Thus, an upper bound for the average number of replacements is $N_R^{\max} = (n/k)/(\alpha + 1)$. A worst-case pattern for replacement events is such that every element of a subsequence of n/s accesses makes reference to the same location, then another subsequence of n/s accesses makes reference to another location, and so on. There are s such subsequences, s/k of them map to the same set (on average), and replacement takes place at every $\alpha + 1$ transitions between them. Thus, a lower bound for the average number of replacements is $N_R^{\min} = (s/k)/(\alpha + 1)$.

Thus, we can estimate the number of replacement-induced transitions as *proportional* to the average number of operations per set, i.e., n/k (best case), or locations per set, i.e., s/k (worst case). The Driver tries to either maximize or minimize that number for stimulating either replacement-induced or collision-induced transitions. Note that, although the model is transition-oriented, it just serves as a *proxy* for (indirectly) increasing coverage under whichever metric is adopted by the design environment. Algorithm 6 relies on such mechanism for setting a distinct constraint on each random test so as to favor either replacements or collisions.

We let x be a Boolean value, and we let RTG(n, s, k) denote the invocation of the RTG engine for creating a test.

The outer loop (lines 2–19) iteratively visits the points corresponding to a given candidate solution G until all are visited or full coverage is reached. In each iteration, P^* keeps the generation subspace $S \times K$ induced by the points of the candidate solution that have not been visited yet.

The inner loop (lines 4–18) visits those points in an order that tries to maximize coverage in minimal time. In each iteration, lines 5–12 induce an alternation between a test favoring replacements and a test favoring collisions, according to the above model. Besides, some ranking is used to better control the effect of alternation. Lines 6 and 7 rank the choice of k before s to stimulate replacement at as most hierarchical levels as possible, whereas lines 10 and 11 rank the choice of k after s to avoid replacement at as most levels as possible, thereby stimulating collisions at most of them. Once a pair (s^*, k^*) is chosen, the inner loop selects the point (n, s^*, k^*) corresponding to the minimum test size (line 13). This selection is another attempt to increase coverage in minimal time. The inner loop induces the generation of a test in each iteration (line 14), and removes the selected point from those to be visited (line 15).

As a result, successive iterations of the inner loop induce tests where the intended alternation is applied to points (n, s^*, k^*) with *possibly distinct* test sizes, as opposed to our early technique [13], whose heuristics restricted exploration to neighborhoods corresponding to *fixed* test sizes. Thus, Algorithm 6 is a generalization that improves the synergy between Driver and Explorer.

Algorithm 6 Driver(G)

```
1:  $x \leftarrow 0$
2:  repeat
3:      $P^* \leftarrow \{(s, k) : \exists (n, s, k) \in G\}$
4:      repeat
5:          if $x = 0$ then
6:              $k^* \leftarrow \min\{k : (s, k) \in P^*\}$
7:              $s^* \leftarrow \max\{s : (s, k) \in P^* \wedge k = k^*\}$
8:              $x \leftarrow 1$
9:          else
10:             $s^* \leftarrow \min\{s : (s, k) \in P^*\}$
11:             $k^* \leftarrow \max\{k : (s, k) \in P^* \wedge s = s^*\}$
12:             $x \leftarrow 0$
13:         $n \leftarrow \min\{n : (n, s, k) \in G \wedge s = s^* \wedge k = k^*\}$
14:         RTG$(n, s^*, k^*)$
15:         $G \leftarrow G \setminus \{(n, s^*, k^*)\}$
16:         $P^* \leftarrow P^* \setminus \{(s^*, k^*)\}$
17:         $V \leftarrow V \cup \{(n, s^*, k^*)\}$
18:     until $(P^* = \emptyset) \vee (cv(V) = 1)$
19: until $(G = \emptyset) \vee (cv(V) = 1)$
```

Finally, let us assess the impact of the proposed approach on time complexity. Test generation is dominated by the time complexity of the RTG engine, which is polynomial [17]. The complexity of the verification algorithms underlying the adopted MCM checker is also polynomial [9].

VI. Experimental Evaluation

A. Experimental Set Up

We evaluated the hybrid test generator (HTG) built with the proposed approach as compared to pure *model-based* and *data-driven* generators. The former is the coverage-driven test generator (CTG) built with our early technique [13]. The latter is the state-of-the-art McVerSi [7] test generator (MTG), which is available in the public domain [18]. We preserved all genetic parameters exactly as they were originally set in [7].

We relied on a checker similar to [9] and on gem5's infrastructure [19] for simulation (*O3*, *Ruby*, and *simple* as CPU, memory, and network models). For the designs, we adopted either a 2-level (L1, L2) or a 3-level (L0, L1, L2)[5] MESI directory protocol with 4-KB (directed-mapped) private caches at L0, 64-KB (2-way) private caches at L1, and a 2-MB (8-way) shared L2 cache, all with same block size (64 bytes).

For observing to which extent the generators are independent of coverage metric, they were all evaluated under two different metrics: one tracking *structural* coverage, another tracking *functional* coverage. While running a test, we tracked the number of distinct transitions covered in the code of each cache controller's FSM. For the structural metric, no distinction was made between identical controller instances in distinct core domains (as in [7]). However, a functional metric should distinguish memory events in multiple core domains as a result of every coherence transaction. Since the global product of all local FSMs leads to a state space that grows exponentially

[5]This is the original labeling used in the gem5 implementation, which conceptually corresponds to the standard levels (L1, L2, L3).

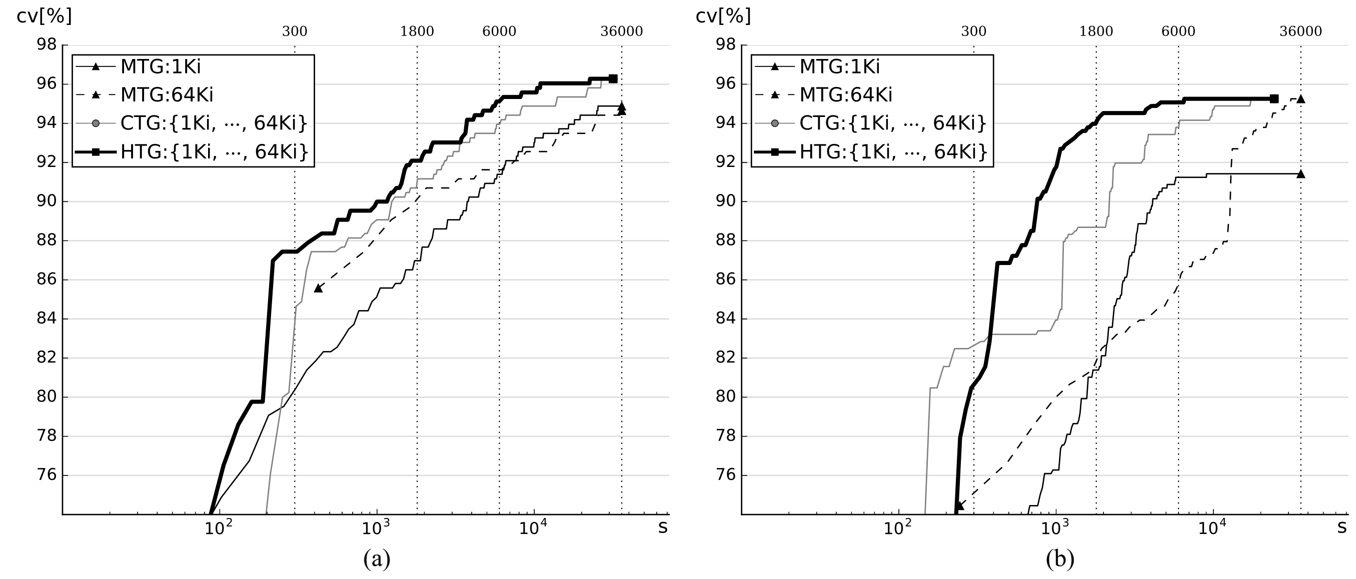


Fig. 4. Structural coverage evolution for 32-core designs as a function of the required time (limited to 10 h). (a) MESI 3-level. (b) MOESI 2-level.

TABLE II
Studied Errors for MESI 3-Level Designs

ID	State	Input event	Next state	Precluded output action
E1 (L1)	IS_I	Data_all_Acks	I	writeDataFromL2Response
	IS	Inv	IS instead of IS_I	(preserved)
E2 (L1)	SM	Data_all_Acks	M	(preserved as in (IM, M))
	SM	Data	SM	(preserved as in (IM, SM))
E3 (L1)	IS_I	DataS_fromL1	I	writeDataFromL2Response
E4 (L1)	E_IL0	WriteBack	MM_IL0	writeDataFromL0Request
E5 (L1)	S	L0_Invalidate_Own	SS instead of S_IL0	forward_eviction_to_L0

TABLE III
Studied Errors for MOESI 2-Level Designs

ID	State	Input event	Next state	Precluded output action
e1 (L1)	SI, OI, MI	Writeback_Ack_Data	I	Data block in sendData
e2 (L2)	ILXW	L1_WBDIRTYDATA	M	writeDataToCache
e3 (L1)	OM	Fwd_GETS	OM	Data block in sendData
e4 (L2)	ILOXW	L1_WBCLEANDATA	M	writeDataToCache
e5 (L1)	SM	Fwd_GETS	SM	Data block in sendData

with core count, a useful metric should avoid full enumeration by restricting the scope while still capturing the impact of the most relevant memory events. That is why we adopted a functional metric that distinguishes transitions between identical controller instances in distinct core domains.

The generators differ in which parameters are *statically* defined by the user and which are *dynamically* set by their engines. The MTG requires the static definition of a *fixed* test length and a *single* address-space constraint, while the HTG and CTG dynamically exploit *variable* test lengths and *multiple* address-space constraints (as a result of varying the parameter k). We report results for the MTG under the (best) static constraint defined in [7]: the test memory size TM = 8 KB. McVersi is free to select as many locations as available within its useful address space. For HTG and CTG, however, the number of locations must belong to a prespecified range. To accommodate such difference, we compared the generators under the same upper bound on the number of (useful) memory locations. Under the adopted constraint that enforces uniform distribution of locations over cache sets, the maximum number of block-aligned locations is 128 TM = 8 KB. As a result, we adopted $S = \{4, 8, 16, 32, 64, 128\}$ for HTG and CTG.

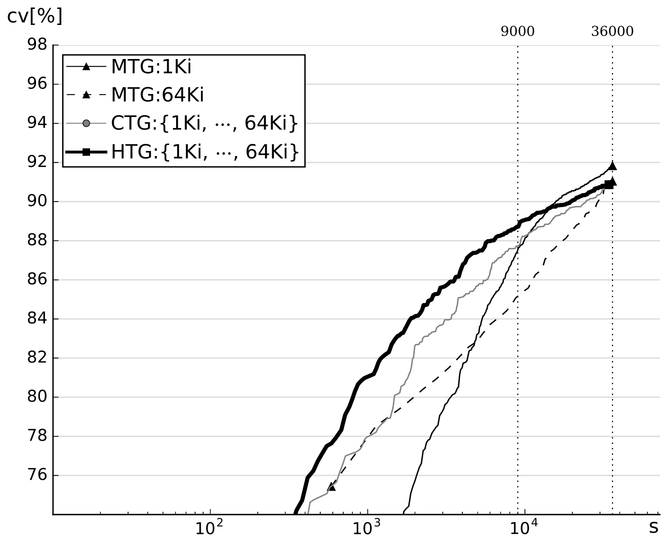
We first measured the cumulative coverage resulting from the execution of a sequence of tests on designs containing *no errors*. Then we inserted different artificial errors by changing the FSMs (either by modifying the next state or precluding some due output action).[6] Each faulty design contained a single, distinct error. The errors studied in our experiments are described in Tables II and III. We measured the runtime until the error was found or until the Directing Engine stopped generation. Each generated test was executed five times under different simulation states (not related to the test itself) in such a way that the distinct executions of the same test are all perturbed differently [7]. To get the reported values, we launched each generator ten times with different seeds, and we took the median values of the resulting distributions. Runtimes were measured in an Intel Xeon E5430 workstation (2.66 GHz, 8-GB memory).

B. Structural Coverage Evolution

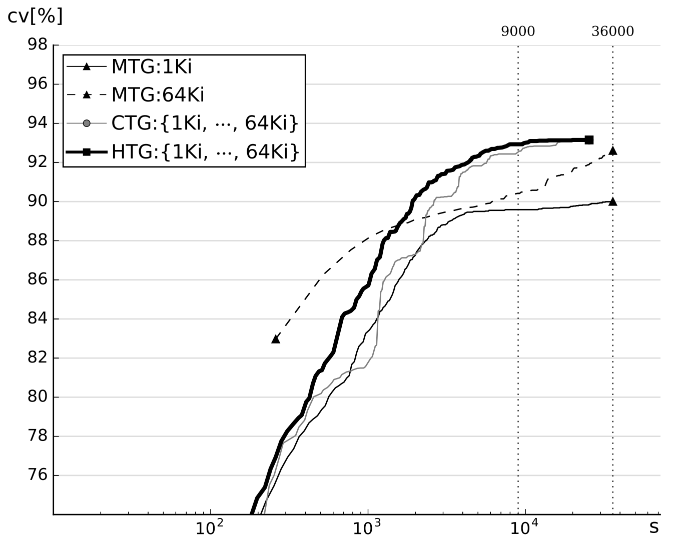
Since the initial verification phase does not display interesting aspects of the generators under comparison (due to transitions that are easy to stimulate randomly), we focus on the intermediate phase (after a few tests have been executed) and the final phase (after many tests have been executed), where sophisticated techniques are needed for increasing coverage.

Fig. 4(a) shows the evolution for an MESI 3-level design. Let us first focus on the MTG's behavior. In the intermediate

[6]Although our checker is also able to find consistency errors, we focused on coherence errors for experimental convenience, without loss of generality.



(a)



(b)

Fig. 5. Functional coverage evolution for 32-core designs as a function of the required time (limited to 10 h). (a) MESI 3-level. (b) MOESI 2-level.

phase, large tests pay off. As compared to $n = 1$ Ki, tests with $n = 64$ Ki lead to higher coverage (until they break even around 6000 s). In the final phase, short tests pay off instead, because increasing coverage values (above a certain threshold) becomes more dependent on the MTG's genetic algorithm, which is sensitive to test throughput. As a result, for covering 94.65% of the transitions, the MTG required 7 h when $n = 1$ Ki, but around 10 h when $n = 6$ 4Ki.

Let us now compare the CTG's behavior with the MTG's. Since the CTG is not data-driven, it stops generation as soon as the generation space is exhausted (in this case, after 9 h). As opposed to the MTG, the CTG automatically adjusts the test size during verification. The CTG became superior to the MTG after running for 5 min (for $n = 1$ Ki) and 30 min (for $n = 64$ Ki). After that, the CTG always led to higher coverage. The MTG reached its maximal coverage (94.88%) after 7 h, whereas the CTG reached that same coverage after only 2.5 h, and attained its own maximal coverage (96.28%) after 7.5 h.

Finally, let us compare the HTG with the others. HTG and MTG reached their highest coverage values, i.e., 96.28% and 94.88% in around 6 and 7 h, respectively. Although both allow dynamic coverage control, the HTG reached the MTG's maximal coverage 4.6 times faster. This indicates that the HTG's exploration of neighborhoods and exploitation of constraints (when properly coupled) can lead to better evolution than genetic-based learning. Albeit both HTG and CTG dynamically exploit constraints, the former exhibited superior coverage evolution from the intermediate up to the final phase. Thus, the superiority of the HTG over the CTG does not come from their similar model-based policy (serialization of constraints), but it is due to the HTG's data-driven policy (exploration of neighborhoods). The CTG favors test throughput: it *fully* explores a plane of the generation space corresponding to the minimum test size, before fully exploring the next plane corresponding to a larger test size. The results indicate that the HTG's policy is better for two reasons: 1) the *partial* exploration of a plane of generation space allows the early run of a few larger test programs, which favors coverage growth and 2) the perturbation function (devoid of built-in heuristics) is more effective for exploring multiple test sizes in the same suite.

Fig. 4(b) shows that the more complex MOESI 2-level protocol enhances the differences between the generators. The MTG's behavior was more sensitive to test size: it is able to achieve the same final coverage as the HTG only for $n = 64$ Ki. The CTG was always superior to the MTG (after the three initial minutes). The HTG reached the maximal coverage 4.8 times faster than the MTG, although both exploit dynamic coverage control. Besides, the HTG stopped generation after 7 h (when the whole generation space was explored), while the MTG kept generating tests up to the 10-h time limit. After 7 min, the HTG became superior to the CTG until they reach the same coverage (95.26%), which clearly shows the impact of the novel data-driven Explorer.

The contrast between Fig. 4(a) and (b) indicate that designs relying on more complex protocols (such as MOESI) are likelier to benefit from the novel data-driven Explorer and from the generalized model-based Driver, because their synergy leads to higher controllability.

C. Functional Coverage Evolution

Fig. 5(a) shows the functional coverage evolution for an MESI 3-level design. Similar to what happened for the structural metric, short tests pay off for the MTG at the final verification phase. After 2.5 h, HTG, CTG, and MTG covered, respectively, 88.72%, 87.73%, and 87.53% of the transitions. After 10 h, both HTG and CTG ended up reaching the same coverage of 90.96%, while the MTG reached 91.04% for $n = 64$Ki and 91.84% for $n = 1$ Ki. This is the only scenario where higher final coverage was observed for the MTG as compared to the HTG. The most likely difference to explain this behavior seems the MTG's exploitation of finer granularity for the amount of shared locations.

TABLE IV
TIME FOR FINDING ERRORS IN MOESI 2-LEVEL 32-CORE DESIGNS

		MTG		CTG	HTG
Error	Metric	1Ki	64Ki	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}
E1	structural	138 (10)	57 (10)	109 (10)	**29** (10)
	functional	89 (10)	72 (10)	114 (10)	**20** (10)
E2	structural	622 (10)	166 (10)	114 (10)	**69** (10)
	functional	827 (10)	160 (10)	126 (10)	**75** (10)
E3	structural	524 (10)	493 (10)	352 (10)	**43** (10)
	functional	919 (10)	798 (10)	362 (10)	**61** (10)
E4	structural	7370 (8)	2752 (10)	**261** (10)	665 (10)
	functional	5491 (9)	2649 (10)	**690** (10)	838 (10)
E5	structural	11958 (10)	17532 (8)	2026 (10)	**821** (10)
	functional	11110 (10)	23980 (8)	1769 (10)	**851** (10)

TABLE V
TIME FOR FINDING ERRORS IN MOESI 2-LEVEL 32-CORE DESIGNS

		MTG		CTG	HTG
Error	Metric	1Ki	64Ki	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}
e1	structural	8 (10)	48 (10)	8 (10)	**7** (10)
	functional	8 (10)	47 (10)	8 (10)	**7** (10)
e2	structural	8 (10)	47 (10)	8 (10)	**7** (10)
	functional	8 (10)	48 (10)	8 (10)	**7** (10)
e3	structural	49 (10)	48 (10)	78 (10)	**35** (10)
	functional	42 (10)	51 (10)	75 (10)	**25** (10)
e4	structural	2692 (5)	2643 (10)	1041 (10)	**641** (10)
	functional	4191 (5)	2552 (10)	**1043** (10)	1073 (10)
e5	structural	3863 (3)	2861 (10)	982 (10)	**586** (10)
	functional	4011 (2)	1965 (10)	1050 (10)	**1044** (10)

Fig. 5(b) shows the functional coverage evolution for an MOESI 2-level design. Similarly to what was observed for the structural metric, the MTG reached higher coverage (92.62%) for the largest test size than for the shortest one (90.01%). However, even after running for 10 h, the MTG covered less transitions (92.62%) than the HTG was able to cover (93.15%) in only 5.5 h. This contrasts with the *structural* coverage evolution observed for MOESI 2-level, where the MTG reached the same final coverage as the HTG. This indicates that the HTG can reach higher coverage than the MTG, because the higher complexity of the MOESI protocol is better captured by the more expressive metric. Similarly to what was observed for the structural metric, the HTG displayed better evolution than the CTG, but the same final coverage (93.15%).

By contrasting all four coverage evolution scenarios, we conclude that our hybrid approach was superior in most cases, especially, for the more complex MOESI protocol. This indicates that a well-designed hybrid approach is likely to cope with the growing complexity of protocols used in high-end designs, which is better tracked by more expressive coverage metrics.

D. Error Discovery Rate and Detection Time

Tables IV and V show the median time (in seconds) required by each technique for exposing every error. They show the best values in bold and report (between parentheses) how many (out of ten) test suites exposed an error.

For MESI 3-level designs, all CTG and HTG suites exposed every error (while some suites MTG suites failed to expose E4 and E5). The HTG required the least time in all cases, except for E4. However, in that case, the MTG took 1.5–2 h to uncover the error, which was found by the HTG in only 10–14 min. The HTG was from 1.65 to 8 times faster than the CTG (except for E4, for which the CTG was 2.5 times faster). The HTG was from 1.65 to 28 times faster than the MTG.

For MOESI 2-level designs, the HTG required the least time in nine out of ten cases, where it was from 1.14 to 3 times faster than CTG, and it was from 1.14 to 6.6 times faster than the MTG. The exception was e4 under the functional metric, for which the CTG was slightly faster than the HTG (in this case, the CTG's greedy shorter-test-size-first heuristic happened to pay off). The HTG and the CTG were able to expose *every* error in less than 18 min, while the MTG took as much as 1 h and 10 min. Besides, the MTG (for n=1Ki) was unable of exposing errors in e4 and e5 for 50% and 70%–80% of the test suites.

Overall, the HTG was able to consistently expose every error in *all* trials, independently of metric and protocol, while the MTG's error discovery rate was sensitive to metric and test size. Thus, the HTG seems more robust to cope with different verification scenarios.

VII. CONCLUSION

Our experimental evaluation relied on 54 483 test runs, each one executed five times in distinct simulation conditions, i.e., 272 415 test executions in total. The results show that hybrid coverage-driven RTG can be superior to purely data-driven or model-based approaches, not as a result of plain composition, but due to proper *decoupling* of competences: data-driven (neighborhood) exploration and model-based (constraint) exploitation. The results also indicate that a well-designed hybrid approach is likely to pay off as designs tend to rely on more complex protocols, being effective in discovering design errors (quite independently of the coverage metric adopted in a given design environment). By allowing superior coverage evolution and high error discovery rate, a hybrid approach can raise the confidence in the face of tight verification deadlines.

REFERENCES

[1] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, Jun. 2012.

[2] S. Devadas, "Toward a coherent multicore memory model," *Computer*, vol. 46, no. 10, pp. 30–31, 2013.

[3] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[4] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," *Comput. Aided Verification*, vol. 6174, pp. 258–272, Dec. 2010. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-14295-6_25

[5] J. Alglave *et al.*, "GPU concurrency: Weak behaviours and programming assumptions," in *Proc. ACM Int. Conf. Architect. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2015, pp. 577–591.

[6] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated synthesis of comprehensive memory model litmus test suites," in *Proc. ACM Int. Conf. Architect. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2017, pp. 661–675.

[7] M. Elver and V. Nagarajan, "McVerSi: A test generation framework for fast memory consistency verification in simulation," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect. (HPCA)*, 2016, pp. 618–630.

[8] O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, and M. Horowitz, "Verification of chip multiprocessor memory systems using a relaxed scoreboard," in *Proc. 41st IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, 2008, pp. 294–305.

[9] L. S. Freitas, E. A. Rambo, and L. C. V. dos Santos, "On-the-fly verification of memory consistency with concurrent relaxed scoreboards," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2013, pp. 631–636.

[10] X. Qin and P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2012, pp. 3–8.

[11] I. Wagner and V. Bertacco, "MCjammer: Adaptive verification for multi-core designs," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2008, pp. 670–675.

[12] Y. Lyu, X. Qin, M. M. Chen, and P. Mishra, "Directed test generation for validation of cache coherence protocols," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 1, pp. 163–176, Jan. 2019.

[13] G. A. G. Andrade, M. Graf, N. Pfeifer, and L. C. V. dos Santos, "Steep coverage-ascent directed test generation for shared-memory verification of multicore chips," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2018, pp. 29:1–29:8.

[14] S. Fine, A. Freund, I. Jaeger, Y. Mansour, Y. Naveh, and A. Ziv, "Harnessing machine learning to improve the success rate of stimuli generation," *IEEE Trans. Comput.*, vol. 55, no. 11, pp. 1344–1355, Nov. 2006.

[15] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A program for verifying memory systems using the memory consistency model," *ACM SIGARCH Comput. Architect. News*, vol. 32, no. 2, pp. 114–123, 2004.

[16] K. Gharachorloo, "Memory consistency models for shared-memory multiprocessors," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, USA, 1995.

[17] G. A. G. Andrade, M. Graf, and L. C. V. dos Santos, "Chaining and biasing: Test generation techniques for shared-memory verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, doi: 10.1109/TCAD.2019.2894376.

[18] M. Elver. (2016). *Mcversi Framework*. [Online]. Available: https://github.com/melver/mc2lib

[19] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Architect. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

Gabriel A. G. Andrade received the B.Sc. and M.Sc. degrees in computer science from the Federal University of Santa Catarina, Florianópolis, Brazil, in 2014 and 2016, respectively, where he is currently pursuing the Ph.D. degree in automation and systems engineering.

His research focuses on algorithms for test generators and runtime checkers targeting multicore chip verification.

Marleson Graf received the B.Sc. degree in computer science from the Federal University of Santa Catarina, Florianópolis, Brazil, in 2017, where he is currently pursuing the M.Sc. degree in computer science.

His research focuses on algorithms for test generators and runtime checkers targeting multicore chip verification.

Nícolas Pfeifer received the B.Sc. degree in computer science from the Federal University of Santa Catarina, Florianópolis, Brazil, in 2018, where he is currently pursuing the M.Sc. degree in computer science.

His research focuses on algorithms for test generators and runtime checkers targeting multicore chip verification.

Luiz C. V. dos Santos received the Doctoral degree from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1998.

He was with the Design Automation Section of the Information and Communication Systems Group, Eindhoven University of Technology. He is a Professor with the Federal University of Santa Catarina, Florianópolis, Brazil. His current research interests include algorithms for verification and test of multicore chips.