Online Periodic Test of Reconfigurable Scan Networks

Lylina, Natalia; Wang, Chih-Hao; Wunderlich, Hans-Joachim

Proceedings of the IEEE Asian Test Symposium; Taichung, Taiwan; Nov. 2022

doi: https://doi.org/10.1109/ATS56056.2022.00026

Abstract: Reconfigurable Scan Networks (RSNs) access embedded instruments throughout the whole system lifecycle. To support dependability management by means of RSNs, RSNs themselves must be continuously tested. The paper-at-hand presents the first online periodic test method for RSNs. The developed algorithm generates a short sequence of test patterns, which tests all parts of an RSN. The generated sequence is uploaded on-chip and is applied periodically to avoid fault accumulation in RSNs. The overall test application time is minimized to comply with the timing requirements of the well-known safety standards. The experimental results show that the method is efficient for all considered RSN designs and is scalable with the increasing size and complexity of RSNs.

Preprint

Online Periodic Test of Reconfigurable Scan Networks

Natalia Lylina, Chih-Hao Wang, Hans-Joachim Wunderlich
ITI, University of Stuttgart, Pfaffenwaldring 47, D-70569 Stuttgart, Germany
{lylina, wangco, wu}@informatik.uni-stuttgart.de

***Abstract*—Reconfigurable Scan Networks (RSNs) access embedded instruments throughout the whole system lifecycle. To support dependability management by means of RSNs, RSNs themselves must be continuously tested.**

The paper-at-hand presents the first online periodic test method for RSNs. The developed algorithm generates a short sequence of test patterns, which tests all parts of an RSN. The generated sequence is uploaded on-chip and is applied periodically to avoid fault accumulation in RSNs. The overall test application time is minimized to comply with the timing requirements of the well-known safety standards. The experimental results show that the method is efficient for all considered RSN designs and is scalable with the increasing size and complexity of RSNs.

***Keywords*-Reconfigurable Scan Networks, design-for-test, online test, periodic test, test generation**

I. Introduction

Modern digital systems integrate an extensive number of instruments, which enhance the system dependability. Reconfigurable Scan Networks (RSNs), as standardized by IEEE Std. 1149.1 [1] and IEEE Std. 1687 [2], provide an efficient access to the instruments. In Fig. 1, each instrument is accessed in parallel through a scan segment. The values in the control segments cs_0, cs_1, and cs_2 determine a dynamically activated Active Scan Path through the scan segments, as shown with a yellow color. In the provided example, the activated path traverses the control registers, as well as the scan segments s_1 and s_3. For better readability, only the selected instruments are shown in Fig. 1.

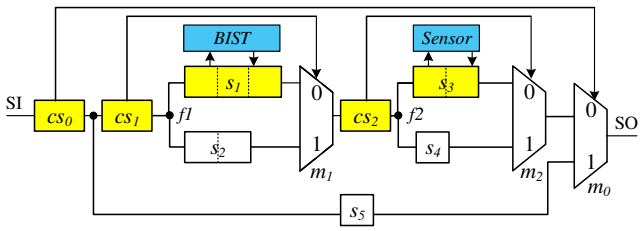

Fig. 1. An example of a Reconfigurable Scan Network

Nowadays RSNs are increasingly used in-field to support silicon lifecycle management. The data collected by RSNs from sensors, monitors, and Built-In Self-Test (BIST) registers is continously analyzed for ensuring dependability, safety and even security [3–7]. Runtime-adaptive instruments, such as adaptive voltage and frequency scaling (AVFS), error rate adoption or temperature control blocks, can be accessed by RSNs, and RSNs can be used to control online BIST registers [8]. Reliability threats, such as faults or abnormal changes in the system behavior [3–5, 7] can be fetched by RSNs from monitors and sensors. In the case of performance degradation, RSN-based fault-handling mechanisms can be used to adjust the operating conditions and thereby to prolong the useful lifetime of a device [6].

The applications above depend on the correct operation of RSNs. Since RSNs occupy a significant fraction of the chip area, the probability of a fault therein is not negligible. Conventional methods [9–13] test RSNs once, after the manufacturing, or perform online concurrent test of RSNs [14]. To avoid fault accumulation concurrent test has to be complemented by periodic test, since components not used for some time may be still subject to aging [15]. The need for periodic test is especially strong in safety critical applications like automotive [16, 17].

The paper-at-hand presents the first online periodic test method for RSNs. The goal of the presented method is to access all primitives of an RSN with a minimized number of selected activated scan paths. A short scheduled set of access sequences is generated based on the selected scan paths, stored on-chip and applied to test an RSN online. The overall test application time is minimized to comply with the functional safety requirements.

The remainder of this paper is organized as follows. First, in Section II, background information about RSNs and the existing test methods is provided. Section III provides a top-down overview of the presented test generation method. In Section IV, a scheduling approach is provided, which activates those ASPs which to minimize the overall test application time. In Section V, it is shown how to generate a minimized set of active scan paths, which cover all the components of an RSN. The experimental results in Section VI show the efficiency of the presented approach and its applicability for a wide set of benchmarks. Section VII concludes the paper.

II. Background

This section presents background information about RSNs and the considered fault models. The latter part of the section summarizes the existing methods to test RSNs and highlights their limitations.

A. Reconfigurable Scan Networks

Each scan segment (s_1 to s_5 in Fig. 1) of an RSN consists of one or multiple scan cells, and each scan cell includes a shift flip-flop and an optional shadow flip-flop.

Control scan segments (cs_0 to cs_2) issue control signal for control primitives, such as multiplexers (m_0 to m_2) and Segment Insertion Bits (SIBs). Depending on the value of the address control signal, a scan multiplexer includes one of the incoming branches into an activated path. A SIB either includes or excludes an underlying segment into a path. Such a path, which can be activated by valid assignments to address

control signals through an RSN, and which traverses the RSN from a primary scan-input (SI) to a primary scan output (SO) is further referred to as an Active Scan Path. The scan primitives include the scan segments and the control primitives.

A scan configuration c in an RSN is defined as the state of the scan primitives. In a valid scan configuration, only one scan path through selected scan primitives is activated. The set C defines all valid scan configurations.

Each access through an RSN can be represented as a single Capture-Shift-Update (CSU) operation [18]. During a capture-phase, the data from the instruments is captured into the scan segments. During the shift-phase, the data is shifted through an activated path towards the scan-output, while new data is shifted-in. This data can come from an Automated Test Equipment or from the cloud, or can even be stored on-chip. Finally, during the update-phase, the data from the shadow flip-flops of the scan segments is latched into the instruments or is used to propagate the control signals from the control scan segments to the control primitives. The *transition relation* of an RSN defines such pairs of scan configurations, which can be reached from one another within one CSU-operation.

An RSN is modeled as a graph $G := (V, E)$, as shown in Fig. 2. Each vertex $v_i \in V$ models a scan segment, a multiplexer input or output, a scan-in/out port. The vertices are annotated with their address control signals.

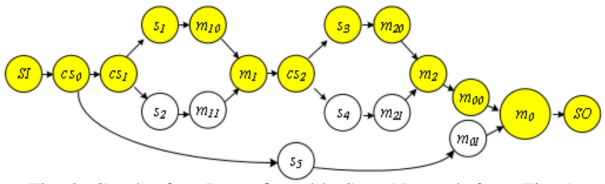

Fig. 2. Graph of an Reconfigurable Scan Network from Fig. 1

An active scan path $asp_j := \{v_0, \ldots, v_i, v_{i+1}, v_k\}$ is an ordered sequence of vertices, such that an edge $(v_i, v_{i+1}) \in E$ exists in the RSN graph for any pair of vertices (v_i, v_{i+1}) $v_i, v_{i+1} \in asp_j$ and the control signal assignments allow to include both vertices to the path. For a path asp_j, the vertex set $V(asp_j) \subset V$ includes the vertices which belong to this path. The initial state $c_0 \in C$ corresponds to the initial active scan path asp_0, and includes the vertices $V(asp_0) \subset V$.

An access to an instrument of an RSN might require the sequential activation of multiple scan paths. A sequence of paths $Seq_k := \{asp_0^k, asp_1^k, \ldots asp_i^k\}$ starts from the initial state asp_0, which is obtained by applying a global reset.

B. Faults in RSNs

Faults in RSNs might arise in different scan primitives, such as scan segments and control primitives.

Faults in scan segments include two types:

- *Shifting faults*: Fault affecting shift flip-flops might prevent correct data from being latched while shifting. Few important examples here are setup- and hold-time violations, which affect the timing behavior of a scan cell. A shifting fault may affect the integrity of the active scan paths traversing it.
- *Faults at the interfaces to the instruments*: Faults at the capture- and update-circuitry of a scan cells may affect reading and writing data to and from the instruments.

Fault in control primitives such as multiplexers and SIBs, make certain parts of an RSN inaccessible for observation and control. If a scan multiplexer is affected by a stuck-at-X fault, then its X-branch is always selected independent of the current assignment to the address control signal. All the other branches of the multiplexer become inaccessible. A SIB affected by a stuck-at-asserted fault always includes the underlying segment into an active scan path. If it is stuck-at-de-asserted, then the segment is never included.

C. Test of RSNs

Conventional scan chains are tested by shifting-in a flush test sequence, such as "01100" or "10011", and observing the shifted-out sequence at the scan-output. Testing an RSN is more challenging, since only selected primitives are tested. The method from [19] can be used to ensure that single faults are detectable. To test an RSN, a set of test sequences is generated, such that each scan primitive of interest is tested at least once. Each test sequence in the set contains the values which are shifted into an RSN. Two types of sequences exist:

- *Test access sequences* are used to load the control scan segments of the RSN with the required control values to bring the RSN into the desired scan configuration.
- *Test workload sequences* are used to test the scan primitives located on a configured scan path.

If possible, access merging techniques, as in [18], are applied to minimize the test access time.

1) Scan Segments: To propagate a fault effect from a scan segment to the output of the RSN, an scan path must be configured through the target scan segment [12]. Then a flush test sequence is shifted through the active scan path and observed at the scan-output. If the sequence is unchanged, all the segments on the path are considered fault-free, otherwise some scan segment is faulty.

2) Control Primitives: To test a multiplexer, a sequence of paths is configured, such that each scan-input of the multiplexer is included into at least one path. A flush test sequence is shifted through each such path and the output sequences for all these paths are compared to the expected outputs [20]. To test a SIB, a pair of paths is configured such that the first one includes the underlying segment and the second one excludes this segment from the path. If the lengths of the output sequences are different and as expected for both activated paths, the SIB is fault-free.

III. Overview of an Online Periodic Test Generation Method

Due to stringent real-time operation and performance requirements, periodic test must be performed within a limited time frame. For the major time, the system must operate in a functional mode, as shown in Fig. 3. If the periodic test time budget is not sufficient for executing the complete test, it is usually divided into multiple sessions, each within the budget and applied successively [16, 17].

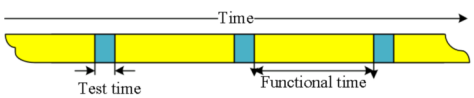

Fig. 3. Periodic test

The test method presented below generates a short set of test access sequences, which are stored on-chip and are applied to

the RSN periodically. Through a set of sequences, the RSN is configured in a way that all scan primitives, which include the scan segments and the inputs of scan multiplexers, are covered within a minimized test time. Thereby, the control signals are also implicitly covered. The test workload sequences can be generated by the existing methods [9–13]. This section formulates the test sequence generation problem in terms of graphs and presents a top-down overview of the solution.

A. Problem Formulation

The solution to the scheduling problem is a minimized set of test access sequences $Test := \{Seq_0, \dots Seq_k \dots Seq_N\}$ for $k = 1, \dots N$, such that all the vertices of the RSN graph are covered at least once and the overall test application time is minimized. The following constraints must hold:

- Each vertex is covered at least once:

$$\bigcup_{Seq_j \in Test} \bigcup_{asp_i^j \in Seq_j} V(asp_i^j) = V \tag{1}$$

- The cost of the test sequence set is minimized:

$$\sum_{Seq_j \in Test} \sum_{asp_i^j \in Seq_j} cost(asp_i^j, Seq_j) \rightarrow min \tag{2}$$

where $cost(asp_i^j, Seq_j)$ is the cost of adding a path asp_i^j at the end of a sequence Seq_j.
For Seq_j and asp_i^j, the cost is calculated as follows:

$$cost(asp_i^j, Seq_j) := switch(asp_i^j, Seq_j) + 2 + |asp_i^j| \tag{3}$$

where $switch(asp_i^j, Seq_j)$ represents the number of cycles which are necessary to configure the path asp_j from the last path in the sequence Seq_j; 2 cycles are required to perform capture and update phases, and $|asp_i^j|$ cycles are requred to perform shift phase.

B. Top-Down Overview

It is possible to obtain an optimal solution with respect to the constraints above, if the computing runtime and the storage capacities are not limited. However, in this case, all possible transitions between the scan configurations must be compared exhaustively, which is not feasible even for medium-sized RSNs. To efficiently generate the test sequence set, an efficient heuristic is presented:

- If all possible transitions between the scan configurations would be considered which are reachable by applying an unlimited number of CSU-operations, an optimum solution would be obtained. However, it is infeasible to explore the resulting solution space even for medium-sized RSNs. To reduce the solution space, at each step, we only consider such pairs of scan configurations (c_1, c_2) that the second configuration c_2 is reachable from the first configuration c_1 by applying a given limited number of CSU-operations.
- In an RSN, the number of configurations may grow exponentially in terms of the number of configuration bits. Therefore, exploiting all possible transitions between the scan configurations is infeasible, even if the number of considered CSU-operations is restricted. In Section V, it is shown how to select those active scan paths, whose activation would bring the highest additional gain among all other candidates with respect to the coverage, without exhaustively checking all the candidates.

IV. Scheduling of Test Accesses

A. Vertex Covering Problem Formulation

A test access scheduling method identifies a set of active scan paths, which cover all the reachable scan primitives in the RSN, and also their activation order. To comply with safety requirements, the test sequence set is minimized with respect to the overall test application time. Let the set ASP include all active scan paths asp_i^j, which are included into the test sequence set at a given time.

The algorithm is applied to the RSN graph, where each vertex is annotated with two values:

- $cost(v_j, asp_i^j)$ shows the additional cost of including the vertex v_j into a path asp_i^j and is defined as the length of the corresponding scan primitive. The costs of all the included vertices in the path asp_i^j are summed up to calculate the length of the path asp_i^j.
- $gain(v_j, ASP)$ represents the gain of covering the vertex v_j. The gain equals to 1, if the vertex has not been previously included into any active scan path in ASP. After a vertex is included at least into one active scan path, its gain is reset to 0.

The transitions between the activated configurations are represented by using the transition relation. Since storing the complete transition relation is inpractical even for medium-sized RSNs, only rather small parts of the transition relation are generated dynamically and stored in the local memory. The transition relation for the running example is shown in Fig. 4. Each bit in a configuration corresponds to a scan multiplexer. The bit is set to 0, if the 0-branch of the multiplexer is selected and to 1 otherwise. Each path can be reset to the initial path asp_0 within 1 CSU-operation, and the reset transitions are omitted in the figure.

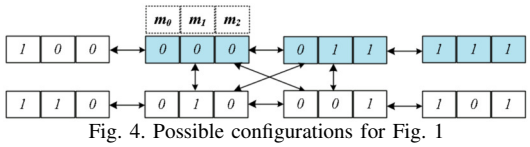

Fig. 4. Possible configurations for Fig. 1

Each vertex of the transition relation graph corresponds to a single path. The edges of the graph show the reachability between the corresponding vertices within one CSU-operation.

At each iteration it of the algorithm, an intermediate test sequence set T_{it} is updated and some new paths are added. At each point, it is possible to assess the cost and the gain of adding a path asp_i^j into the sequence set T_{it}:

- The cost of adding a path asp_i^j at the end of a sequence Seq_j is referred to as $cost(asp_i^j, Seq_j)$ and is defined as in Eq. 3.
- The gain of adding asp_i^j into the sequence set T_{it} is referred to as $gain(asp_i^j, T_{it})$. It is defined as a number of vertices in the RSN graph, which have not been covered by the set T_{it}, but are covered by the path asp_i^j.

Algorithm 1: *generateTestSequenceSet*

Input: RSN graph $G := (V, E)$, where the initial path asp_0 is activated
Output: Test as a set of sequences $Test := \{Seq_0, \dots Seq_k\}$, where $Seq_j := \{asp_0^j, asp_1^j, \dots asp_i^j\}$; Boolean flag $CoveredStatus$ which shows, whether all the requested vertices are covered

```
/* Initialize the initial state asp_0; the first
   path sequence Seq_0; and the test set T        */
asp_0 ← reset;
Seq_0 ← (asp_0)
T_0 ← {Seq_0}
/* Initialize the covered vertices with the
   vertices of asp_0                              */
V_cov ← V(asp_0)
/* asp_0 is included into each path Seq_j so it is
   not needed to cover it explicitly              */
V(asp_0).resetGain()
/* Initialize the distance value                  */
maxDistance ← m
/* Initialize the iterator                        */
it ← 0
/* Proceed until all the vertices are covered     */
while (V_cov ≠ V) do
    k ← k + 1
    /* For each sequence in the test set          */
    for Seq_j ∈ T_it do
        /* Find the candidate paths with a minimum
           distance from Seq_j and relevant positive
           gain                                   */
        candidates ← getPathsMaxGainMinCost(Seq_j)
        /* Select the path to add: if multiple
           branch-and-bound                       */
        addedPath(Seq_j) ← selectPath(candidates)
    end
    /* Collect the potentially added paths        */
    AddedPaths ← ⋃_{Seq_j}^{Test} addedPath_j
    /* If it is not possible to find at least one
       path within the specified distance from any
       of the sequences then stop the operation   */
    if AddedPaths == ∅ then
        break
    end
    /* Select such a sequence S in T_it and a path
       asp, which have the shortest distance
       between them                               */
    (S, asp) ← getTheSequenceToAppend(T_it, AddedPaths)
    /* Add the path at the end of the selected
       sequence                                   */
    S' ← S + asp
    /* Update the coverage                        */
    V_cov ← V_cov ∪ V(asp)
    /* Reset the gain of the included vertices    */
    V(asp).resetGain()
    /* Update the test set with the selected
       sequence and ensure that the reset sequence
       is still in the test set                   */
    T_it+1 ← T_it \ {S} ∪ {S'} ∪ {Seq_0}
    it ← it + 1
end
/* After all the sequences are generated, return
   the final test set and the coverage status     */
Test ← T_it
return (Test, V_cov)
```

B. Test Sequence Generation

Algorithm 1 presents the general scheduling flow:

- *(Line 1-5)*: The computation starts with initializing the initial test sequence set T_0 with the first sequence Seq_0 and adding the ASP asp_0 into the sequence Seq_0.
- *(Line 6)*: The maximum number of CSU-operations for test sequence generation is initialized. Depending on the selected value, the trade-off between the runtime and the test application time is established.
- *(Line 8-24)*: The test sequence generation runs until either all the vertices of the RSN graph are visited at least once or it is not possible to cover any more vertices (Line 16).
- *(Line 10-13)*: For each sequence in T_{it}, possible paths are determined, which are reachable within a specified number of CSU-operations, and which allow to cover more vertices in the RSN graph ($gain(asp_i^j, T_{it}) > 0$). If multiple such paths exist, the added path is selected by using a branch-and-bound approach.
- *(Line 14-17)*: If none of the sequences in the test set can be extended by a path, the test generation converges.
- *(Line 18-21)*: The sequence Seq_j to augment with an additional path asp_i^j is selected, such that the cost of adding the path into the sequence is minimized ($cost(asp_i^j, Seq_j) \rightarrow min$).
- *(Line 22)*: The test sequence set is updated. The selected sequence is augmented with the selected path. It is explicitly ensured that a basic sequence, which only includes the reset ASP, is still in the set.
- *(Line 26)*: The algorithm provides the test sequence set $Test$ and the covered vertices as an output.

Example: *In Fig. 4, just one CSU-operation is considered at a time ($m = 1$). The computation starts at the reset configuration $\{m_0, m_1, m_2\} = \{0, 0, 0\}$. The configurations $\{1, 0, 0\}, \{0, 1, 0\}, \{0, 0, 1\}$ and $\{0, 1, 1\}$ are reachable. The configuration $\{0, 1, 1\}$ has the highest gain and is added into the sequence. At the next step, it is possible either to add a configuration $\{1, 1, 1\}$ into the existing sequence, or to initialize a new sequence, which starts with a reset configuration and includes also one the reachable configurations. The configuration $\{1, 1, 1\}$ has the highest gain and is selected for adding into the existing sequence. All the vertices of the RSN graph are covered with three configurations, which all belong to the same sequence.*

V. Selection of Relevant Active Scan Paths

Given an initial path asp_i^j, it is not required to consider all the paths, which are reachable from this path within a given number of CSU-operations. Instead, it is possible to generate a relevant subset of paths for asp_i^j by mutating the corresponding scan configuration as much as possible within a given number of CSU-operations.

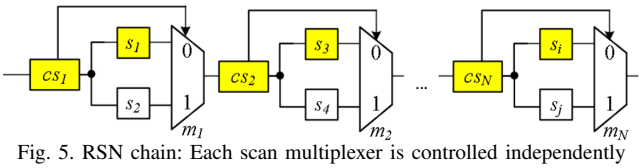

Fig. 5. RSN chain: Each scan multiplexer is controlled independently

This section provides a method to quickly identify those paths, which might have the highest gain among all other candidates. First, an ASP selection method is presented for so-called series-parallel RSNs [19]. Then we show, how the presented method is applied even for those RSNs, which are do not have a series-parallel property.

A. SP-RSNs

Definition 1: Let $G := (V, E)$ be a directed acyclic graph with the vertex set V and the edge set $E \subset V^2$ with a single source $sc \in V$ and a single sink $si \in V$. G is called series-parallel (SP), if one of the following holds:

- $V := \{sc, si\}$
- G is a parallel composition of two series-parallel graphs $G_1 := (V_1, E_1)$, $G_2 := (V_2, E_2)$:

$$V := V_1 \cup V_2; E := E_1 \cup E_2 \tag{4}$$

$$\begin{aligned} &sc := sc_1 = sc_2; si := si_1 = si_2 \\ &V_1 \cap V_2 = \{sc, si\} \end{aligned} \tag{5}$$

  sc_j and si_j are sources and sinks of G_j; $j = 1, 2$.
- G is a series composition of two SP graphs, Eq. 4 holds:

$$sc := sc_1; si := si_2; si_1 = sc_2 \tag{6}$$

Any directed graph, which does not fulfill the conditions above is referred to as a *non-series-parallel graph.*

B. SP-RSN Path Selection

For SP-RSN graphs, the hierarchical dependencies between the graph vertices are stored in a binary decomposition tree, as shown in Fig. 6. The "P" vertices show the parallel connections between the vertices, while the "S" vertices represent the vertices connected in series.

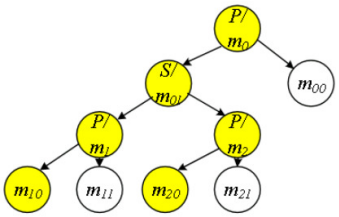

Fig. 6. Dependencies between the multiplexers for the RSN graph in Fig. 2

Analyzing this tree has a logarithmic complexity and is used to identify which vertices belong to the same path according to the RSN topology. A valid path in an RSN is defined from the top vertex of the tree by traversing the tree in a depth-first-search manner. Each time, when a "P" vertex is met, only one branch must be selected for further investigations. If a "S" vertex is met, both branches must be selected. To generate the most relevant scan configurations from a given path asp_i^j within 1 CSU-operation, the states of the multiplexer vertices are flipped starting from the lowest-level vertices in a reserse polish order, until the first "P" vertex is met. If more CSU-operations are considered, the flipping is also allowed for the second lower-level multiplexer vertices.

Example: *In Fig. 6, the initial path is shown with yellow. The next path is obtained by flipping the states of the vertices* m_1 *and* m_2*. The next path traverses the lower branches of* m_1 *and* m_2 *through the vertices* s_2 *and* s_4*.*

C. Path Selection for non-SP-RSNs

It is straight-forward to check, if an RSN is series-parallel [19]. According to our experiments, most of the available benchmarks are already series-parallel. A general RSN structure can be transformed into a series-parallel representation by inserting a small number of virtual vertices.

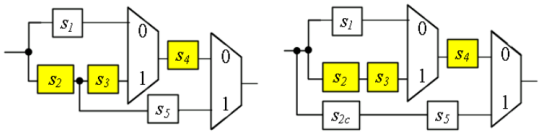

Fig. 7. Non-series-parallel RSN and its transformation

The newly added vertices (such as s_2 and s_{2c} in Fig. 7.c) are referred to as twins. Any pair of twins is mutually equivalent for the means of testing, which means that if one of them is tested by a given sequence - another one is tested as well. Applying the procedure above for series-parallel models of non-series-parallel RSNs is a pessimistic estimation - the test pattern sequence, which is generated for a series-parallel representation, covers all the primitives in the original RSN but may require longer test application time, since the twins might be tested more than one time.

VI. Experimental Results

In the experiments, the test access sequence sets have been generated according to the developed method. The solution space has been explored and a tradeoff between the test cost and the coverage of scan primitives has been investigated. The coverage of scan primitives is defined as a fraction of scan primitives, which are accessed by a given test sequence set, from the whole set of scan primitives. A set of close to pareto-optimal solutions has been generated by means of genetic algorithms. For a required coverage, the best generated solution with respect to test cost has been selected.

The genotype describes a test sequence set as a sequence of Boolean values, where each sequence comes after another, and each sequence contains one or multiple configurations. Each bit shows the state of a control scan primitive in the configurations. The initial genotypes are created following Algorithm 1. Each next configuration is chosen so that the gain is maximized as shown in Section V. The choice is randomized within the most prominent candidates to efficiently explore relevant parts of the search space. The crossover operation (with a probability of 0.95) between two-parent genotypes creates two child genotypes from two parental genotypes. The first child inherits the first part of test sequences from the first parent and the second part from the second parent, for the second child the inheritance is reversed. The mutation operation (with a probability of 0.05) randomly removes one sequence from the test sequence set or adds a new sequence into the set.

All the experiments have been performed using an Intel(R) Xeon(R) W-2125 CPU at 4.00GHz with 132 GB of main memory and implemented in the eda1687 tool of [18]. The RSN benchmarks have been taken from the widely-recognized ITC'2016 [21] benchmark set. The basic characteristics of the benchmarks are provided in Table I, and includes the information about the design name (Column 1), the number of multiplexers, SIBs, scan segments and scan cells in Column 2, 3, 4 and 5 correspondingly.

Table II shows the experimental results. The genetic programming solver Opt4J [22] is used with the NSGA-II method [23] to perform the optimization. The number of generations to find a solution is given in Column 2 and is determined as a minimum number of generations when the solutions stop improving for 5 consequent operations. The size of the initial population is given in Column 3. The time to generate the solution is provided in Column 4, which is rather low for all the benchmarks. Column 5 shows the number of required test sequences and Column 6 represents the total number of configurations. For all the benchmarks, the generated test sequence set covers all the scan primitives. The test cost is given in Column 7. Given a minimal required coverage of

TABLE I. Benchmarks Characteristics

(1)**Design**	(2) #muxes	(3) #sibs	(4) #segs	(5) #cells
BasicSCB	10	-	21	176
Mingle	13	10	22	270
TreeFlat	24	12	24	101
TreeUnbalanced	28	28	63	41,887
TreeBalanced	46	43	90	5,581
TreeFlat_Ex	60	57	123	5,194
q12710	25	25	47	26,183
a586710	47	-	79	41,682
p34392	142	-	245	23,261
t512505	160	-	288	77,006
p22810	283	283	537	30,111
p93791	653	-	1,241	98,637

scan primitives, e.g. 90%, the genetic algorithm generates a test sequence set with a significantly decreased test cost, as provided in Column 8.

The experimental results show that the developed method is efficient for a wide range of RSN designs and is scalable with an increasing size of RSNs. Scalable graph-based modeling and efficient mapping to a genetic programming problem instance allow to generate a test sequence set with a required coverage, while minimizing the test cost. Since the test sequences are independent of one another they can be applied individually, during independent online periodic test sessions to meet safety margins.

VII. Conclusion

The paper presents the first online periodic test of RSNs, which ensures that all critical scan primitives are examined within a safety interval with a minimized number of test patterns. The generated test sequences can be used throughout the whole RSN lifetime, both as an online periodic test compliant with the safety requirements and as an efficient offline manufacturing test to reduce the test application time on ATE, The experimental results show the scalability and the efficiency of the presented method for all the considered RSN sizes and topologies.

Acknowledgments

This work was supported by the German Research Foundation (DFG) under grant WU 245/17-2 (ACCESS) and partially supported by Advantest as part of the Graduate School "Intelligent Methods for Test and Reliability" (GSIMTR) at the University of Stuttgart. We thank the group of Prof. Teich for providing access to the Opt4J framework.

Bibliography

[1] "IEEE Standard for Test Access Port and Boundary-Scan Architecture," *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444, May 2013.

[2] "IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device," *IEEE Std 1687-2014*, pp. 1–283, Dec. 2014.

[3] A. M. Y. Ibrahim and H. G. Kerkhoff, "An On-chip IEEE 1687 Network Controller for Reliability and Functional Safety Management of System-on-Chips," in *Proc. Int'l. Test Conf. in Asia (ITC-Asia)*, 2019, pp. 109–114.

[4] F. G. Zadegan, D. Nikolov, and E. Larsson, "A self-reconfiguring IEEE 1687 network for fault monitoring," in *Proc. European Test Symp. (ETS)*, 2016, pp. 1–6.

[5] A. Tsertov, A. Jutman, K. Shibin, and S. Devadze, "IEEE 1687 Compliant Ecosystem for Embedded Instrumentation Access and In-Field Health Monitoring," in *Proc. IEEE AUTOTESTCON*, 2018, pp. 1–9.

[6] K. Shibin, S. Devadze, and A. Jutman, "On-line fault classification and handling in IEEE1687 based fault management system for complex SoCs," in *Proc. Latin-American Test Symp. (LATS)*, 2016, pp. 69–74.

[7] M. A. Kochte and H.-J. Wunderlich, "Self-Test and Diagnosis for Self-Aware Systems," *IEEE Design & Test*, vol. 35, no. 5, pp. 7–18, 2018.

[8] P. Raiola, B. Thiemann, J. Burchard, A. Atteya, N. Lylina, H.-J. Wunderlich, B. Becker, and M. Sauer, "On Secure Data Flow in Reconfigurable Scan Networks," in *Proc. Conf. on Design, Automation Test in Europe (DATE)*, Mar. 2019, pp. 1–6.

[9] D. Ull, M. Kochte, and H.-J. Wunderlich, "Structure-Oriented Test of Reconfigurable Scan Networks," in *Proc. IEEE Asian Test Symp. (ATS)*, Nov. 2017, pp. 127–132.

[10] M. A. Kochte, R. Baranowski, M. Schaal, and H. Wunderlich, "Test Strategies for Reconfigurable Scan Networks," in *Proc. Asian Test Symp. (ATS)*, 2016, pp. 113–118.

[11] R. Cantoro, A. Damljanovic, M. S. Reorda, and G. Squillero, "A Novel Sequence Generation Approach to Diagnose Faults in Reconfigurable Scan Networks," *IEEE Trans. on Computers*, vol. 69, no. 1, pp. 87–98, 2020.

[12] R. Cantoro, L. San Paolo, M. Sonza Reorda, and G. Squillero, "An Evolutionary Technique for Reducing the Duration of Reconfigurable Scan Network Test," in *Proc. IEEE Int.-l Symp. on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, Apr. 2018, pp. 129–134.

[13] P. Habiby, S. Huhn, and R. Drechsler, "Power-aware Test Scheduling for IEEE 1687 Networks with Multiple Power Domains," in *Proc. IEEE Int.-l Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2020, pp. 1–6.

[14] C.-H. Wang, N. Lylina, A. Atteya, T.-Y. Hsieh, and H.-J. Wunderlich, "Concurrent Test of Reconfigurable Scan Networks for Self-Aware Systems," in *Proc. IEEE Int.-l Symp. on On-Line Testing And Robust System Design (IOLTS)*, Virtual, Jun. 2021, pp. 1–7.

[15] C. Liu, E. Schneider, and H.-J. Wunderlich, "Using Programmable Delay Monitors for Wear-Out and Early Life Failure Prediction," in *Proc. ACM/IEEE Conference on Design, Automation Test in Europe (DATE'20)*, 2020, pp. 1–6.

[16] B. Kaczmarek, G. Mrugalski, N. Mukherjee, A. Pogiel, J. Rajski, L. Rybak, and J. Tyszer, "LBIST for Automotive ICs with Enhanced Test Generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pp. 1–1, 2021.

[17] C. Eychenne and Y. Zorian, "An effective functional safety infrastructure for system-on-chips," in *Proc. IEEE Int.-l Symp. on On-Line Testing and Robust System Design (IOLTS)*, 2017, pp. 63–66.

[18] R. Baranowski, M. A. Kochte, and H.-J. Wunderlich, "Reconfigurable Scan Networks: Modeling, Verification, and Optimal Pattern Generation," *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 2, pp. 1 – 30, 2015.

[19] N. Lylina, C.-H. Wang, and H.-J. Wunderlich, "Testability-Enhancing Resynthesis of Reconfigurable Scan Networks," in *Proc. of the IEEE Int.-l Test Conf.(ITC)*, Virtual, Oct. 2021, pp. 1–10.

[20] R. Cantoro, F. G. Zadegan, M. Palena, P. Pasini, E. Larsson, and M. S. Reorda, "Test of Reconfigurable Modules in Scan Networks," *IEEE Trans. on Computers (TC)*, vol. 67, no. 12, pp. 1806–1817, 2018.

[21] A. Tsertov, A. Jutman, S. Devadze, M. S. Reorda, E. Larsson, F. G. Zadegan, R. Cantoro, M. Montazeri, and R. Krenz-Baath, "A suite of IEEE 1687 benchmark networks," in *Proc. IEEE Int'l Test Conf. (ITC)*, Nov. 2016, pp. 1–10.

[22] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4J - A Modular Framework for Meta-heuristic Optimization," in *Proc. Genetic and Evolutionary Computing Conf. (GECCO)*, Jul. 2011, pp. 1723–1730.

[23] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

TABLE II. Periodic Test of Reconfigurable Scan Networks

	NSGA-II			Sequence Optimization results		Coverage 100%	Coverage 90%
(1)**Design**	(2) # generations	(3) # population	(4) runtime [m:s]	(5) # sequences	(6) # configurations	(7) cost	(8) cost
BasicSCB	10	100	00:32	1	7	265	102
Mingle	30	300	02:11	1	11	385	199
TreeFlat	10	100	00:02	2	4	3,890	2,506
TreeUnbalanced	30	300	02:16	7	39	66,416	28,464
TreeBalanced	30	300	05:02	2	9	57,610	31,747
TreeFlat_Ex	30	300	04:11	2	9	24,226	11,078
q12710	10	300	01:12	1	4	39,394	30,118
a586710	30	300	04:30	1	7	55,957	34,376
p34392	30	500	10:11	3	2	40,923	20,410
t512505	50	1000	05:16	2	6	143,112	102,557
p22810	100	1000	10:10	9	41	379,373	210,214
p93791	100	1000	15:13	4	28	1,225,512	469,840