FSWGEN: a Device-tree Specification driven System-Level Test workload generator

FSWGEN: a Device-tree Specification driven System-Level Test workload generator

Gabriele Filipponi
Dip. di Automatica e Informatica, Politecnico di Torino, Turin, Italy
gabriele.filipponi at polito.it

***Abstract*—This paper explores the portability and reusability of functional System-Level Test (SLT) methodologies across different System-on-Chip (SoC) design architectures. The proposed approach leverages the Device-tree Specification (DTS) of the Device-Under-Test (DUT) to enable automated test generation. By constructing a graph representation from the DTS, the methodology facilitates the creation of functional test programs across different SoC architectures through a simple graph traversal process, making it highly adaptable and efficient in simplifying SLT development.**

I. Introduction

As System-on-Chip (SoC) designs continue to increase in complexity [1], ensuring their reliability before shipment has become more challenging. Conventional structural tests, which focus primarily on individual components, often fail to cover faults that emerge from the interactions between these components [2]. To bridge this coverage gap in safety critical sectors, like automotive, and meet stringent quality and safety standards such as ISO 26262, System-Level Test (SLT) has been introduced into the manufacturing test flow over the past decade [3]. Despite its advantages, SLT comes with notable challenges. Writing functional test procedures is labor-intensive, error-prone, and tightly coupled to specific hardware architectures and instruction set architectures (ISAs). As a result, its often difficult to port or reuse across different SoC platforms. To address these limitations, this paper proposes a novel framework that automates the generation of functional procedures using the Device-tree Specification (DTS) a hierarchical, representation of the SoC's hardware, including components, interconnections, and capabilities [4]. The DTS provides a unified abstraction of the architecture and ISA, enabling the generation of portable and reusable test code.

II. Background

A. Manufacturing test flow

The manufacturing test flow of safety critical sectors includes multiple stages, such as Wafer Test, Package Test, Burn-In, System-Level Test (SLT), and Final Test to identify faulty devices early [3]. The new step SLT complements structural tests by means of functional procedures, testing the system as a whole, including hardware/software interactions.

B. Device-tree Specification

The Device-tree Specification (DTS) defines a standardized method for describing system hardware using a data structure known as a device-tree. A device-tree is organized as a hierarchical tree, where each node represents a module in the system and contains property/value pairs that define its characteristics and functionalities.

III. Proposed Methodology

This work introduces a novel framework for generating SLT workloads by leveraging the structural and behavioral descriptions defined in the DTS. At the core of the framework is an algorithm that builds and traverses a graph-based representation of the SoC, capturing both connectivity and functional constraints.

A. Graph-Based SoC Modeling

The proposed methodology models the SoC architecture as an undirected graph. Each component is represented as a vertex, and connections between them form the graph's edges. This abstraction enables the representation of both structural hierarchy and inter-connections of the underlying hardware.

Master components (typically CPU cores) are modeled with detailed architectural attributes Figure 1 (a), including registers, architecture sets, and dedicated memory structures such as instruction and data caches. Slave components, Figure 1 (b), (e.g., peripherals) are described in finer granularity, incorporating registers, register fields, and memory blocks.

To further define the behavioral rules of peripherals, (i.e. protected register access sequences or conditional unlocking procedures) custom procedures called *actions* can be defined. These enrich the expressiveness of the model and enable automated handling of complex peripheral behaviors.

In addition, the framework leverages the descriptive and portable nature of the *compatible* and *model* keywords defined in the DTS. These keywords facilitate the incremental loading of device descriptions, allowing hardware components to be matched, reused, and integrated into the SoC graph.

Finally the decoupling of fhe SoC description from individual components improves modularity and promotes high portability, particularly for third-party intellectual property (IP) integration. It also eliminates the need for a monolithic DTS file, supporting modular and reusable hardware descriptions.

B. FSWGen Framework

The framework is based on the algorithm reported in Algorithm 1. It begins by loading the SoC along with all master and slave DTS files. An initial exploration of the master nodes is then performed.

For each master, a matching compatible DTS file is searched for, and if found, it is incrementally integrated into the main graph.

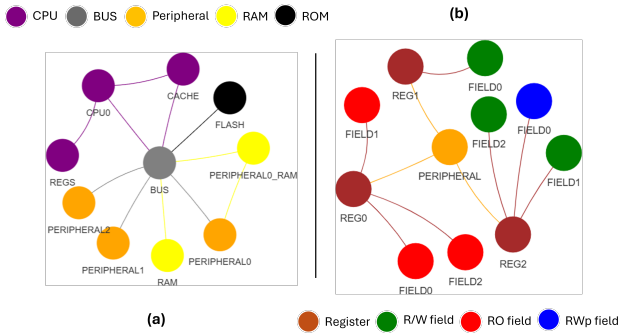

Fig. 1: (a) DTS graph representation for an example SoC and a peripheral (b).

Following this, a second graph traversal is carried out to identify all nodes that are reachable from the master components. For each reachable slave, if a compatible hardware description is found, it is also integrated into the SoC graph.

During the second traversal, functional SLT procedures are generated for each peripheral and master. These procedures, written in assembly code and executed by the master, perform read (R) and write (W) operations on the slave's registers. Furthermore, if a peripheral requires specific actions to unlock certain registers, these actions are automatically carried out according to its action description.

Algorithm 1 Pseudo code of the FSWGEN framework.

Require: $input$, SoC's DTS file.
Require: $cpus\_dir$, cpus include directory.
Require: $peripheral\_dir$, peripherals include directory.
$cpus \leftarrow \{\}$, $peripherals \leftarrow \{\}$
$dts \leftarrow$ load_DTS(input), $graph \leftarrow$ create_graph(dts)
for each $fdts \in scandir(cpus\_dir)$ **do**
  $cdts \leftarrow$ load_DTS(fdts), $cpus[cdts.compatible] \leftarrow cdts$
end for
for each $fdts \in scandir(peripheral\_dir)$ **do**
  $pdts \leftarrow$ load_DTS(fdts), $peripherals[pdts.compatible] \leftarrow pdts$
end for
for each $dcpu \in dts.cpus$ **do**
  if $dcpu.compatible \notin cpus$ **then**
    $break$
  end if
  $cpu \leftarrow cpus[dcpu.compatible]$, $arch \leftarrow$ load_arch(cpu)
  $reach \leftarrow graph.get\_reach(cpu)$
  for each $dperipheral \in reach$ **do**
    if $dperipheral.compatible \notin peripherals$ **then**
      $break$
    end if
    $peripheral \leftarrow peripherals[dperipheral.compatible]$
    $actions \leftarrow peripheral.$ load_actions($include, arch$)
    for each $register \in peripheral.registers$ **do**
      $value \leftarrow 0$
      for each $field \in register.fields$ **do**
        $i \leftarrow (\sim field.rst\_value) \ll field.bit\_pos$
        $value \leftarrow value \mid i$
      end for
      if $register.access \in actions$ **then**
        $actions[register.access](arch, register, value)$
      else
        $arch[register.access](register, value)$
      end if
    end for
  end for
end for

IV. Experimental Results

The proposed framework was successfully used to create functional procedures for a complex SoC, as shown in Figure 2 (a). Table I reports preliminary results of the incremental fault coverage for both Stuck-At (SAF) and Transition Delay (TDF) for the modular controller area network peripheral (CAN), shown in Figure 2 (b), provided by the proposed approach with respect to scan-based approaches. The generated functional procedures, despite their simplicity, results in a substantial increase in coverage, which highlights its effectiveness. Moreover test time is just a tiny fraction of the scan-based test, hence it can be easily included in the test suite by manufacturers.

TABLE I: Incremental fault coverage for the CAN peripheral.

Test Nature	Test Name	Test Time [ms]	SAF [%]		TDF [%]	
			Incr.	Δ	Incr.	Δ
Structural	SCAN	7×10^9	97.89	-	89.38	-
Functional	FSWGEN	1.1	98.61	**0.72**	90.35	**0.97**

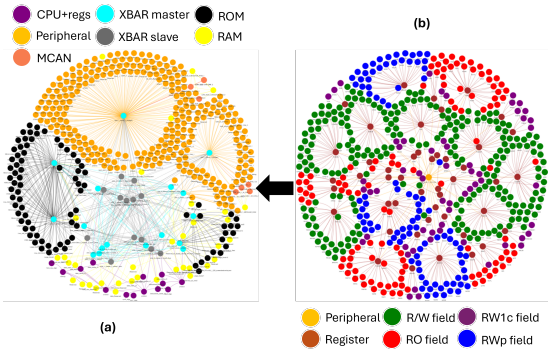

Fig. 2: (a) DTS graph representation for a complex SoC. (b) CAN peripheral DTS graph representation loaded into the SoC's graph.

V. Conclusions & Future Works

In conclusion, this paper presented a framework for generating SLT functional procedures for SoC by leveraging the DTS. The detailed nature of the DTS not only provides a practical foundation for constructing functional routines but also facilitates portability across different architectures and designs by explicitly describing component compatibility.

The core of the framework is a graph-based traversal method that generates R/W operations targeting the slave components of the SoC. As a direction for future work, the framework could be extended to support the specification of functional actions directly within the DTS descriptions of peripherals. This enhancement would enable the automatic generation of more complex functional procedures.

References

[1] A. Vassighi, O. Semenov, M. Sachdev, A. Keshavarzi, and C. Hawkins, "Cmos ic technology scaling and its impact on burn-in," *IEEE Transactions on Device and Materials Reliability*, vol. 4, no. 2, pp. 208–221, 2004.
[2] F. Angione, P. Bernardi, N. d. G. Giardino, G. Filipponi, C. Bertani, and V. Tancorre, "A system-level test methodology for communication peripherals in system-on-chips," *IEEE Transactions on Computers*, vol. 74, no. 2, pp. 731–739, 2025.
[3] I. Polian *et al.*, "Exploring the mysteries of system-level test," in *2020 IEEE 29th Asian Test Symposium (ATS)*, 2020, pp. 1–6.
[4] devicetree.org, "Device-tree specification," 2021, https://app.readthedocs.org/projects/devicetree-specification/downloads/pdf/latest/.