A SYSTEMATIC APPROACH TO SYNTHESIS OF VERIFICATION TEST-SUITES FOR MODULAR SOC DESIGNS

Sudhakar Surendran
sudhaksrs@ti.com
WLS
Texas Instruments India Ltd
Bangalore 560 093, India

Rubin Parekhji
parekhji@ti.com
SoC Design Technology Center
Texas Instruments India Ltd
Bangalore 560 093, India

R. Govindarajan
govind@serc.iisc.ernet.in
Super Computer Edn. And Res. Center
Indian Institute of Science
Bangalore 560 012, India

ABSTRACT

Verification is one of the important stages in designing an SoC (System on Chips) that consumes upto 70% of the design time. In this work, we present a methodology to automatically generate verification test-cases to verify a class of SoCs and also enable re-use of verification resources created from one SoC to another. A prototype implementation for generating the test-cases is also presented.

I. INTRODUCTION

System on Chips (SoCs) are complex designs with heterogeneous modules (CPU, memory, etc.) integrated in them. Verification is one of the important stages in designing an SoC. It is the process of checking whether the transformation from architectural specification to design implementation is correct. Verification involves creating the following resources: (i) a test-plan that identifies the conditions to be verified, (ii) a test-case that generates the stimuli to verify the conditions identified, and (iii) a test-bench that applies the stimuli and monitors the output from the design. Test-cases can be created in a programming language like `C' [1] or in a hardware verification language like Specman's `e' [2]. In our work, we focus on programming language based test-cases. Also, we focus on the class of SoCs that are used for digital signal processing (DSP).

Verification is a time-consuming effort taking about 28% [3] to 70% [4] of the total time spent in designing an SoC. The complex and largely manual nature of verification makes it a time-consuming task. As the number and the complexity of modules used in the SoC increases, the time spent in creating the test-cases also increases. Hence, the test-cases are often built from a library of re-usable components [5]. Yet a significant amount of effort is spent in creating the test-cases manually using these components. Currently the automatic test-case generators are either restricted to generation of simple test-cases or complex test-case generators [6] [7] which require detailed inputs on the configuration of the SoC, the interaction between modules, etc., which are time-consuming and error prone.

In our work, we present a methodology that enables automatic generation of test-cases from re-usable components and with minimum inputs. The main contributions are: (i) identification of abstractions and structures in creation of test-cases, (ii) identification of components of a test-case that are re-usable, (iii) development of a framework that enables capturing these components at module level, independent of the SoC and its variations and (iv) development of another framework to generate non-re-usable components of a test-case using properties captured at module and SoC levels. As an illustration of the methodology, a prototype implementation for generating two specific classes of test-cases, one for verifying memory modules and another for verifying data transfer modules is also presented.

The paper is organized into the following sections: Section II discusses the background required to understand the verification process and reviews the related work from the literature. Section III proposes the methodology to synthesize the test-cases. Sections IV and V discuss the application of the methodology to generate test-cases for verification of memory and data transfer modules. Section VI discusses the prototype implementation of our methodology. Section VII presents a quantitative analysis of generating test-cases using our methodology. Section VIII concludes the paper.

II. BACKGROUND AND RELATED WORK

An SoC is designed in the following stages: architecture definition, design implementation, functional verification, design for test and physical design. Of these stages, functional verification is one of the important and time-consuming stages and is the main focus of this work. The term verification refers to functional verification in this paper.

SoCs that are targeted for DSP are often implemented with an embedded DSP processor. These SoCs are capable of reading data from input sources, processing them using DSP algorithms and writing the processed data to output sinks. This

requirement is met by using a central processing unit (CPU) usually a DSP, a set of (data transfer) modules capable of transferring data in and out of the SoC, memories, control modules to control system resources (like clock, power, etc.), and modules like direct memory access (DMA) controller to transfer data between the memory and the data transfer modules.

Figure 1 shows a generic representation of the SoC. Here *SYS_CTRL_A*, SYS_*CTRL_B* and *EXT_MEM_CTRL* are system control modules, *INT_CTRL* is an interrupt control module, *Memory_A* and *Ext_Mem* are memories, *PERI_A* and *PERI_B* are data transfer modules and *TB_A* and *TB_B* are the test-benches for *PERI_A* and *PERI_B* respectively. The test-benches shown are not a part of the SoC but are present in the verification environment of the SOC. The control path indicates the modules controlled by the system control module e.g., *SYS_CTRL_A* controls *PERI_A*, *PERI_B*, *SYS_CTRL_B* and *Memory_A*.

SoC verification focuses on checking the integration of the modules used. The verification of individual modules is done independently [8, 9] before they are integrated. Though the modules are used in the SoC, the test-cases and their components are not re-used. Also, there is no re-use from one SoC to another though they use the same modules. The non-re-usability is due to the differences in SoC configuration like memory map, DMA event to module event mapping, etc.

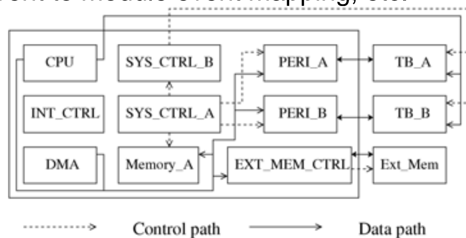

Figure 1: Generic SoC configuration

Integration verification checks whether the SoC formed by integrating these modules operates as per specification. Integration verification happens in the following stages: *basic* (checks read and write access to all the modules in the SoC), *integration* (checks if the module's signals are connected as per specification), *functionality* (checks if the modules integrated into the SoC perform the functions they are expected to perform), *performance* (checks if the performance or bandwidth requirements of the SoC are met) and *application* (checks if the SoC is capable of running the end application).

Emek et al. [6] have proposed a model based test-case generator *X-Gen* designed for SoC verification. *X-Gen* captures the model of the SoC by capturing the types of modules, their properties and configuration in addition to other information. The specific scenarios for which the test-case must be created are also captured. *X-Gen* then generates the test-case using the model of the SoC and the test scenario. *X-Gen* has limited capability to generate expected results for the test-case created.

Cheng et al. [6] have proposed a methodology *SALVEM* (Software Application Level Verification Methodology) that generates test-cases composed of *code snippets*. The *code snippets* initiate and exercise specific operations on an SoC and are associated with a specific module. In addition to the code snippets, the dependency between them and the constraints to generate the parameters for them are also captured. Using these as inputs, *SALVEM* generates the test-case.

In addition to the above, several forms of test-case automation have been known to exist in individual design teams. An illustration is a *basic* stage test-case generator that uses SoC and module configuration captured in a machine-readable format.

To summarize, the existing approaches for automating test-case generation for SoCs are SoC-centric and not module-centric, thus, non-re-useable from module level to SoC level. The work presented in this paper towards the automatic generation of verification test-cases involves identifying the components of the test-case, the functionality and properties of the modules that are independent of the SoC and can be captured at the module level, thereby allowing re-use from module level to SoC level verification and also from one SoC to another. The ability to re-use module resources at SoC level is a significant difference compared to previous approaches. The methodology proposed is generic and can be applied for a wider set of test-cases and SoCs by identifying the appropriate components and properties that must be captured.

III. METHODOLOGY TO GENERATE TEST-CASES

Figure 2 highlights the focus of our work on SoC test-case generation. The contribution of this work is to automate the test-case generation for the highlighted modules. Further, the focus of our verification is on *integration*, *functionality* and *application* checks.

Figure 3 lists the steps followed in our methodology. First, we analyze the test-cases and identify *A*, the steps involved in the execution of the test-case. We then identify *B*, the *components* of a test-case. The *components* are code segments of the test-case that perform an atomic (self-contained)

operation, e.g., code segment to configure the DMA module.

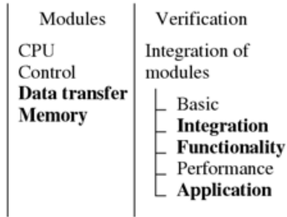

Figure 2: Focus on verification in our work

B consists of two parts: *B1*: Collection of re-usable *components* of the test-case. These components can be captured at module level, (as functions or macros), and can be coded once for the modules during module level verification. They can then be re-used at SoC level and also from one SoC to another.

B2: SoC or test-case specific non-re-usable *components*, i.e., *components* whose contents and existence depend on the SoC configuration and the test-case.

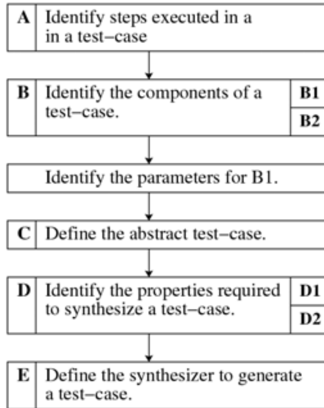

Figure 3: Steps followed in our methodology

In our work, we have structured *B2* such that the *components* therein can be generated automatically. *B2* is made up of *B1* components and other control logic code segments that are specific to the SoC or the test-case. For example, in the code segment for an interrupt service routine, the code segment that checks if an interrupt has been triggered can be captured at the module level. But the interrupt service routine itself is non-re-usable since the interrupt connection can change from one SoC to another.

The next step is to identify the parameters for *B1*. Care is taken to identify the parameters such that they are re-usable and can be generated easily with minimum information. For example, to make the *component* that configures the DMA module re-usable, it should be made independent of any specific DMA architecture. To do this, we have identified the following parameters for the interface: DMA event number, base address of the source or destination module, offset of the memory element in the module, width of the memory element, depth of the memory element, addressing mode for the memory element (FIFO or normal RAM) and endianness. (This is an illustration for DMA module. Similar set of parameters have been identified for other modules as well).

Identifying *B* and its parameters is a medium complex task (based on author's experience) and requires a good understanding of the class of test-case that is intended to be automatically generated.

We then define *C*, the abstract test-case to capture the core functionality performed in the test-case. It is used to generate the actual test-case. Next, we identify *D*, the properties that are captured as *meta-files*, and are required to synthesize: (i) the parameters for *component B1* and (ii) the non-re-usable test-case *component B2*. The properties capture information like the relationship between modules, parameters used by the functions, etc. *D* consists of two parts:

D1: Collection of module level properties. These properties capture the specific characteristics of the module that are required to synthesize the parameters of *B1* and contents of *B2*. Similar to *B1*, *D1* can be created during module level verification and re-used thereafter.

D2: Collection of SoC level properties. These properties capture the specific characteristics of the SoC that differ from one SoC to another and are required to synthesize the parameters of *B1* and contents of *B2*. Though *D2* changes with the SoC configuration D2 is structured such that sections of D2 can either be re-used from other SoC or generated from the architecture specification. The extent of re-use depends on factors like software compatibility, module or sub-system re-use and others. Examples for D2 are base address of modules and DMA event mapping.

The generation of *D* is straight-forward and some sections of *D* can even be made as a part of the SoC and module architecture specification.

The last step in our methodology is defining the synthesizer (*E*) that generates the actual test-case (*F*) using inputs *C*, *D1* and *D2* to drive the test-case generation process, using *components B1* and *B2* in the test-case themselves. Figure 4 presents a high-level overview of the synthesizer and also shows the inputs used and the output generated. *E* internally generates *B2* and then *F* (that uses *B1* and *B2*) using the algorithm which has been identified (which is based on *A*). The generated test-case *F* will contain calls to the *B1 components* captured at the module level.

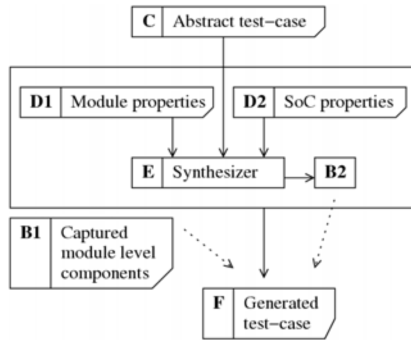

Figure 4: High level overview of the synthesizer

IV. MEMORY MODULE TEST-CASE GENERATION

We analyzed the memory module test-cases and identified the following modules to be active in the test-cases: CPU, system control and memory modules. The component of the test-case that performs the configuration operation can be made re-usable and must be captured at the module level. In addition to this, the component that executes the algorithm to verify the memory module can be made re-usable and hence must be captured as a module level component.

The system control configuration component takes the base address of the module and the properties of the module being controlled as parameters so that they are re-usable. Similarly, the memory verification algorithm components take the properties of the memory module and the module on which the algorithm must be executed as inputs.

The abstract test-case captures the memory modules that must be verified, the memory verification algorithms to be used to verify them and the module on which the algorithms must be run.

The module level properties are captured for memory modules, system control modules, and the memory verification algorithm only. The properties of other modules are not captured since they are not needed to generate parameters or code segments in our methodology.

We propose to capture the following properties at the module level: (i) properties of the memory modules e.g. number of rows and (ii) constraints to generate parameters.

We propose to capture the following properties at the SoC level: (i) base address of the modules, (ii) memory banking details and (iii) the relationship between the system control module and other modules.

The following code segments of the test-case are SoC specific or test-case specific and hence are generated internal to the synthesizer: (i) the parameter of the components captured at module level and (ii) the code sequence to set up all the system control modules in the correct order. The steps used in the synthesizer to generate the test-case are shown in Figure 5.

Step 1. Process SoC and peripheral level meta−files and store the values.

Step 2. Process the abstract test−case and store the details.

Step 3. Generate parameters for memory verification algorithm executed on each memory module.

Step 4. Identify all the system control modules used in the test−case and the order in which to configure them.

Step 5. Generate parameters for the system control module's configuration component.

Step 6. Generate the code for the test−case.

Figure 5: Steps to generate memory module test-case

V. DATA TRANSFER MODULE TEST-CASE GENERATION

We analyzed the data transfer module test-cases and identified that the following modules are active: CPU, DMA control module, interrupt control module, system control modules and data transfer modules. We consider the test-bench in the verification environment as a data transfer module for identifying the components and properties that must be captured.

Figure 6 shows the components of the data transfer test-case that we identified to be captured at module level. The parameter interface for these components abstract out the SoC specific and test-case specific details so that they are re-usable.

The components of system control module take the base address of the module and the properties of the module being controlled as parameters. The components that operate on interrupt events take the properties of the event as inputs like the event number, source event or destination event, etc. The component that configures the DMA module takes as input the parameters discussed in Section III.

Module	Components captured
System control	Configuration
Interrupt control	Configuration, event enable, event status check and, event clear
CPU	Interrupt enable and data transfer
DMA	Configuration, interrupt status check and, interrupt clear
Data transfer and test−bench	Configuration, data generation, module status check, transfer enable, transfer status check, event status check, event clear and, data check

Figure 6: Components captured at module level

The components of the data transfer module take the following parameters as inputs: (i) the base address of the module, (ii) the mode of operation of the module and (iii) the values required for data

transfer e.g., the address to which the transfer must be made.

The abstract test-case captures (i) the data flow in the test-case and (ii) the memory modules to be used as source or destination buffers for the data transfers in the test-case.

The module level properties are captured for data transfer modules and interrupt control modules only. We propose to capture the following properties of the data transfer module at the module level: (i) event properties, (ii) modes of operation supported and (iii) constraints to generate the parameters. For the interrupt control module we propose to capture the mapping of the input interrupt to output interrupt as a property.

We propose to capture the following as properties at SoC level: (i) base address of all the peripherals in the SoC, (ii) interrupt mapping for the interrupt controllers (iii) DMA event mapping for the DMA controllers, (iv) the relation between system control module and other modules and (v) the modes supported by the modules at the SoC level.

The following components of the SoC are test-case specific or SoC specific and hence are automatically generated: the parameters of the components captured, the code segment to set up the system control modules in the required order, the code segment to configure the peripherals that handle interrupts, the interrupt service routine code segment and the code segment that handles data transfer.

The data transfer module test-case synthesizer generates the test-case using the steps shown in Figure 7.

Step 1. Process the meta−files and the abstract test−case and store the values.

Step 2. For each data transfer module in the abstract test−case, generate 3P mode of transfer.

Step 3. For each data transfer module in the data flow, generate parameters for its components.

Step 4. Generate the global values for the test−case.

Step 5. Generate the parameters of the data transfer module that refer to global values.

Step 6. Generate the parameters for the DMA module's components.

Step 7. Identify the connection details of all the interrupts used in the test−case.

Step 8. Identify all the system control modules used and the order in which to configure them.

Step 9. Generate the code for the test−case.

Figure 7: Steps to generate data transfer module test-case

VI. IMPLEMENTATION DETAILS

We have implemented a prototype of the methodology proposed. In our implementation we generate 'C' language test-cases. The components of the modules are captured in 'C' libraries and, the properties of the modules and the SoC are captured as ASCII files. The abstract test-case is also captured as an ASCII file. Perl [10] language is used to implement the test-case synthesizer.

Figure 8 shows the major blocks in our implementation of the test-case synthesizer (TS). The TS consists of two major blocks namely, the subroutine extractor and the synthesizer. The subroutine extractor processes the module level *meta-files* and extracts the parameter generating subroutines, (that capture the constraints to generate the parameters). These subroutines are then added to the synthesizer's subroutine library.

The synthesizer consists of four major blocks: the *meta-file* processor, the abstract test-case processor, the data structure generator and the code generator. Each block generates one or more data structures that are used by the subsequent blocks.

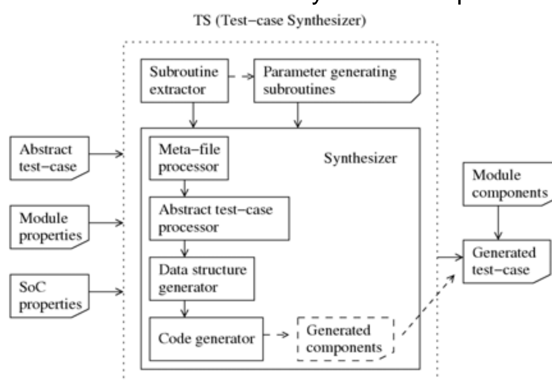

Figure 8: Blocks in test-case synthesizer implementation

VII. QUANTITATIVE ANALYSIS

Figure 9 shows the process involved in creating and executing a test-case manually. Of these steps, significant amount of time is spent on Steps 2, 3, 4 and 8. Step 2 consumes six to twelve person-months (for a representative class of designs based on the SoC verification projects that the author has worked on). But this effort has to be spent once and can be re-used across many SoC projects with minor SoC-specific modifications, the effort for which can be within one to two person-weeks.

The effort spent in Steps 3, 4 and 8 depends on the module functionality and the complexity of the SoC in which the modules are integrated. Also, these steps are specific to the SoC and the module they are targeted. On an average, it takes one to four person-months for Steps 3, 4, and 8 together for generating the test-cases manually per module. This effort varies based on (i) the complexity of the SoC, (ii) the number of modules, (iii) the complexity of the modules, and other such factors.

With the above steps as reference, our methodology for automatic test-case generation impacts Steps 2, 3, 4, and 8. The author spent

approximately five weeks in developing the prototype (discussed in Section VII) to illustrate the benefits of the proposed methodology. The production system that can be used in SoC verification projects will be much more complex than the prototype and is estimated to take about six person-months. But this is a one-time effort and can be re-used across many SoC verification projects (similar to Step 2).

The effort spent in Steps 3 and 4 for automatic generation is five person-days. It is estimated that a similar effort is required for Step 8. Thus the total effort required for automatic generation is ten person-days per module. This is significantly lower when compared to one to four person-months per module spent on manual generation. The benefits accrue further if the components and the properties of the modules are re-used (across multiple SoCs) and the properties of the SoC are captured as a part of the SoC specification.

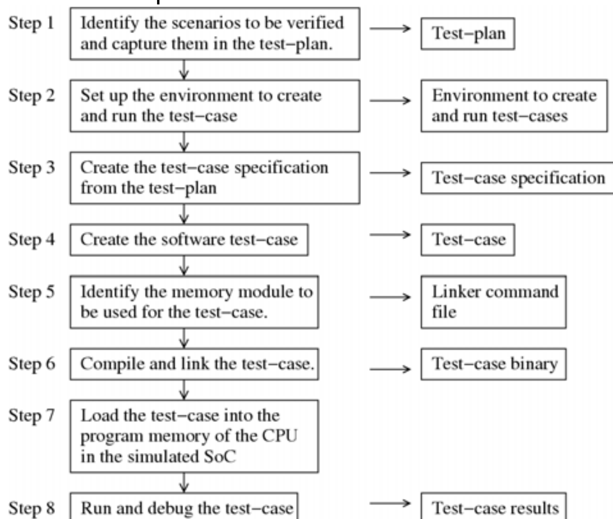

Figure 9: Process involved in creating and executing a test-case

VIII. CONCLUSIONS

In this work, we have presented a systematic approach to synthesize verification test-cases. The approach is based on identification of re-usable components and properties that can be captured at the module-level and the SoC-level. As an illustration the methodology has been implemented to generate memory and data transfer test-cases. The methodology can be extended and applied to other categories of test-cases too. The framework presented in this work for the automatic generation of test-cases further helps to target more efficient and more comprehensive verification of SoC designs.

REFERENCES

1. Brian W. Kernighan and Dennis Ritchie, “C Programming Language”, Pearson Education, second edition, March 1989.
2. Cadence, “e Language Reference”, 2005.
3. United Business Media EETimes online. 2005 Electronic Design Automation Branding Study - Chip Design. http://i.cmpnet.com/eetimes/eda/edasurvey chip.pdf , 2005.
4. 2002 IC/ASIC Functional Verification Study. Technical report, Collett International Research, 2002.
5. F. Hunsinger, S. Francois, and A.A. Jerraya, “Definition of a systematic method for the generation of software test programs allowing the functional verification of system on chip (soc)”, *4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions*, pp 11–16, May 2003.
6. R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, “X-gen: a random test-case generator for systems and socs”, *Seventh IEEE International High-Level Design Validation and Test Workshop*, pp. 145–150, October 2002.
7. A. Cheng, C. Lim, and A. Parashkevov, “A software test program generator for verifying system-on-chips”, *Tenth IEEE International High-Level Design Validation and Test Workshop*, pp. 79–86, December 2005.
8. K. Albin, “Nuts and bolts of core and soc verification”, *38th Design Automation Conference*, pp. 249–252, 2001.
9. W. Yang, M. Chung, and C. Kyung, “Current status and challenges of soc verification for embedded systems market”, *IEEE International [Systems-on-Chip] SOC Conference*, pp. 213–216, September 2003.
10. Larry Wall, Jon Orwant, and Tom Christiansen, “Programming Perl”, O’Reilly Media Incorporated, third edition, July 2000.