Verification Environment Automation from RTL

Zhidong Chen, Yunyang Song, Wenting Hou,
Junna Qiao, Junxia Wang, Ling Bai, Kei-Wang Yiu
Mediatek, Inc.
Email: zhidong.chen@mediatek.com

Abstract-**As the scale of an SoC increases, the verification environment becomes much more complex. For example, the DUT may contain up to hundreds of interfaces. Verification environment automation becomes a must. In this paper, we propose a solution that can automatically build a verification environment from RTL. We use bus interfaces as an example to demonstrate how to extract design information from RTL. To be specific, we have used pattern matching to extract interface signals based on bus protocols. Besides interface signals, other information like data width, read/write ability can also be extracted from RTL. We have compared the execution of 3 projects before and after deploying this solution respectively. It is found that the effort in building the verification environment is reduced by ~70% after deploying this solution.**

I. Introduction

As an SoC scales upwards, the verification environment becomes much more complex. For example, if we want to verify the bus fabrics of the entire SoC, the DUT (Design Under Test) may contain up to hundreds of interfaces. It is prohibitive and error prone to code all the verification IP (VIP) connections by hand. So verification environment automation becomes a must. The design specification (either executable or pure documentary) is a good candidate for automatically building the verification environment. However, the design specification is not always well aligned with RTL changes. Therefore, it is preferable to extract design information directly from the RTL itself to build the verification environment.

Fig. 1 shows a typical SoC bus architecture and the corresponding verification environment. In an SoC, there can be hundreds of bus interfaces under various protocols (ACE, ACE-Lite, AXI, AHB, APB, etc.). For each bus interface, the verification engineers need to gather the following information to build the verification environment:

1. VIP interface connection: the design hierarchy and the signal names in RTL.
2. VIP configuration: the capabilities of this bus interface.
3. VIP stimulus constraint: the type of the transaction or sequence this bus interface can send/ receive.

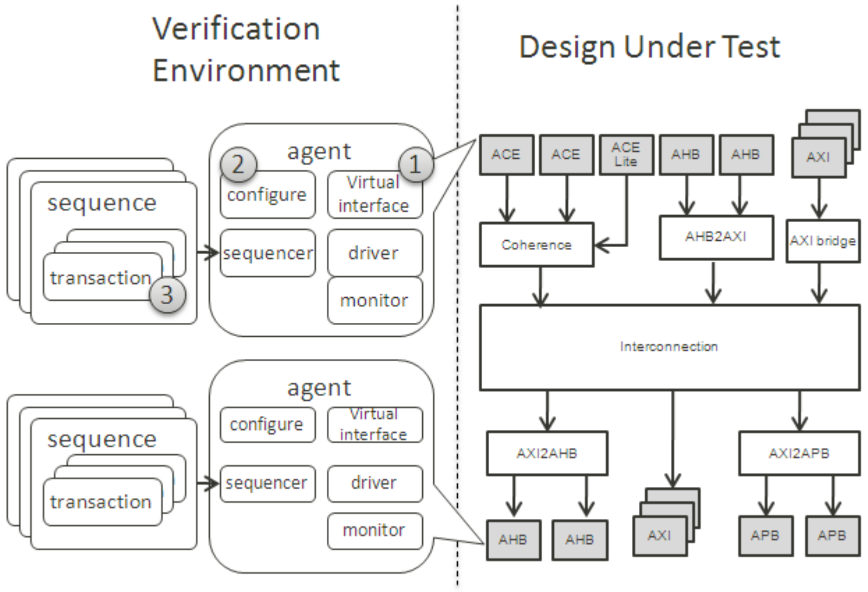


Figure 1. A typical SoC bus architecture and the corresponding verification environment.

Accordingly, the challenges to build the verification environment arise from the following aspects:

1. VIP interface connection
    a. There are many signals in a bus interface. For example, an AXI master has more than 50 signals (including user-defined side-band signals).
    b. For a design with hundreds of interfaces, thousands of signals need to be connected. For example, one of our designs has more than 180 interfaces. The verification engineers need to connect over 4200 interface signals.
2. VIP configuration
    a. A lot of variables need to be configured. For example, there are over 70 configurable variables in our AXI master VIP. Among them, 20 variables are design related. So the verification engineers need to collect the corresponding information from the design team.
    b. For a design with 180 interfaces, more than 1800 variables need to be configured in total.
3. VIP stimulus constraint
    a. Taking timing and area into account, bus components are not always fully functioning. So the verification engineers need to customize the stimulus for each VIP.
    b. For a bus interface of master type, the verification engineers need to know what kind of transactions it can send. For example, an AXI master with only read channels cannot send write transactions.
    c. A bus interface of slave type should avoid receiving transactions beyond its capabilities. Otherwise it may cause false alarms.

Traditionally, all the information is collected manually and the testbench is also hand-coded. Collecting such information takes a lot of time and effort. In addition, there is no auto-checking mechanism. So the mismatches between RTL and the design specifications cannot be found until the simulation fails. Once the RTL is changed, the information and the corresponding testbench will be out of date. It may take even more effort in debugging to find and correct the out-of-date design information. On the other hand, a lot of information as mentioned above can be automatically collected. For example, many of the interface signals and the configurable variables can be extracted from RTL. With necessary information, the testbench (including the stimulus constraints) can also be generated automatically.

Reference [1] introduces a method to automatically instantiate and connect VIPs. It can effectively reduce the effort to generate VIP connections, despite the fact that it still requires users' intervention. However, it does not provide a solid solution to maintain the testbench with changing RTL.

In this paper, we propose a systematic solution to extract design information from RTL, and to build the verification environment automatically from the extracted information. The solution is illustrated in Fig. 2. We will use bus interfaces as an example to demonstrate how to extract design information from RTL. According to our observation, in practice, around 80% interface signals follow the same naming convention. We have used pattern matching to extract those signals based on bus protocols. Besides interface signals, other information like data width, read/write ability can also be extracted from RTL. Also, we provide an easy-to-use GUI for the users to review and revise the extracted information and to input those that cannot be extracted. Moreover, the users' input is checked against RTL to ensure its consistency with RTL and to eliminate human errors. With mature verification libraries at hand, the users can use built-in commands to generate the verification environment automatically. Besides, we also provide some APIs for the users to customize their own scripts.

This solution has the following benefits:

1. The time spent in building the verification environment can be reduced from months to weeks.
2. It is ensured that the VIP connections are well aligned with RTL, thereby relieving DV from tedious debugging work.
3. It is extendable. It can be applied to both standard bus (APB, AHB, AXI, etc.) and user-defined bus. Besides, it can be applied in both VMM and UVM.
4. The verification environment can be easily migrated, either from VMM to UVM, or from one project to another.

This paper is organized as follows. In Section II, we will introduce how to extract information from RTL. Here, we will use bus interfaces as an example. The idea can be also applied to other forms of information. In Section III and IV, we will introduce how to build bus bridge type verification environment for a single module and a sub-system respectively. Finally in Section V, we will present some results after deploying this solution to show its benefits.

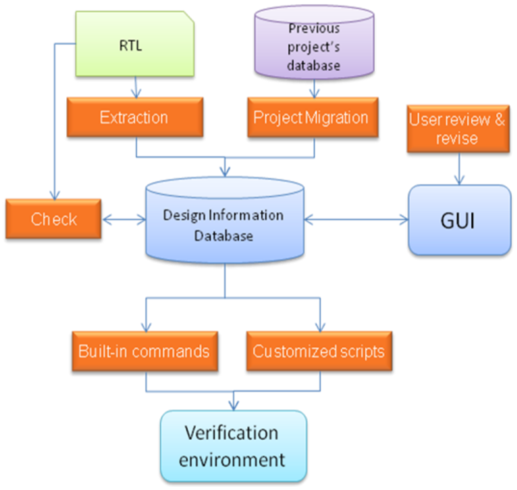


Figure 2. Flow for automatic verification environment generation.

II. INFORMATION EXTRACTION FROM RTL

In this section, we will introduce how to extract design information from RTL for the purpose of building the verification environment. We will use bus interfaces as an example, but the idea can be applied to extraction of other information. After the design information is extracted, the users can validate the correctness of the information, and to input the missing information on a GUI. Then the design information is checked against RTL to ensure their consistency. To further reduce the users' effort, some information from previous designs can be reused. We will illustrate how to reuse this information.

A. *Information Extraction*

1) *Configuration –The Bus Definition*

In order to extract a bus interface, we should know about the definition of this bus first. The job of defining a bus is left open to the users. In other words, we do not (and do not need to) know the definition of this bus beforehand. The users should provide a configuration file explaining the bus interface. Generally speaking, the configuration file contains the following items: bus protocol (e.g., AXI), type (master or slave), protocol signal name, signal direction (input or output), etc., as shown in Table I. In this way, we are able to extract not only standard bus interfaces but also user-defined interfaces, without the specific knowledge of bus protocols. For standard bus interfaces (e.g., AHB, APB, AXI), we can prepare their configuration files beforehand to save the trouble for users. As for user-defined interfaces, the users need to prepare their configuration files, using those of standard interfaces as templates. An example of APB bus configuration is shown in Table I. Here the last column specifies the "seed" signal of this bus protocol. The seed signal is used as a template for name pattern matching during the extraction. (See the next section for more details.)

TABLE I. AN EXAMPLE OF APB BUS CONFIGURATION

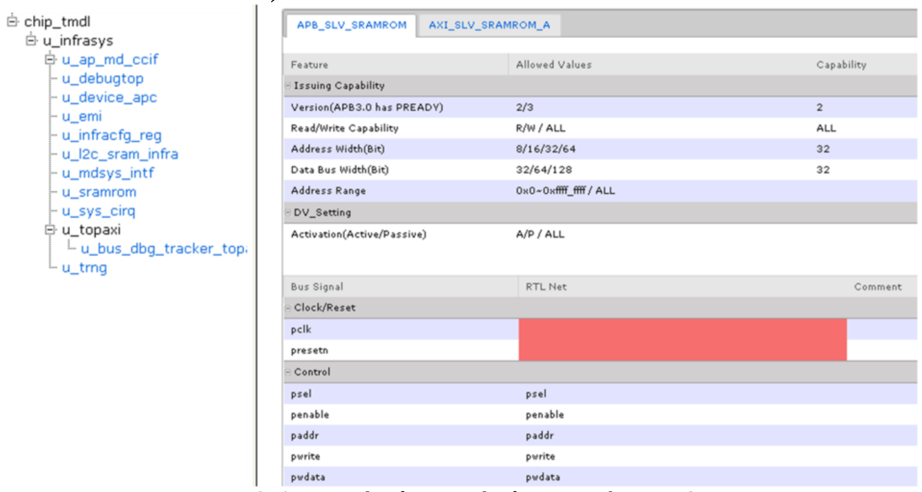
Protocol	Type	Signal	Direction	Seed
APB	master	pclk	input	N
	master	presetn	input	N
	master	penable	output	N
	master	paddr	output	Y
	master	pready	input	N

2) *Bus Interface Extraction*

With the bus protocol configuration, we are then able to extract bus interfaces from RTL. There are several approaches to do the extraction. One approach is to extract all possible interface signals and then group them according to naming patterns. However, this approach would produce many collateral signals that do not belong to any bus interface. Preferably, the user may specify one or several "seed" signals in the configuration file. A seed signal is a bus protocol signal that is obligatory in this protocol and is most likely to appear in the same naming convention with other signals. It is used as a template for name pattern matching. In our implementation, the address signals (e.g., "paddr" in APB as shown in Table I) are always chosen as seed signals, so that the users do not have to worry about the selection of seed signals. Once the seed signals are specified, their naming patterns (e.g., prefix and postfix) are first extracted, and then used to find other signals in the same bus protocol. Using this approach, we are able to extract bus interfaces from the RTL of any instance of interest.

Besides interface signals, other information can be extracted from RTL too, if we know how and where to get this information. Specifically, once a bus interface is extracted from a certain instance, its capabilities (e.g., address width, data width and read/write ability) can be also extracted from the RTL. For example, after bus interface extraction, if an AHB interface is found to exist in an instance, its address width can be obtained from the width of the port connecting to "haddr"; its data width can be obtained from the width of the port connecting to "hrdata" or "hwdata"; its read/write ability can be determined by if this interface has read or write signal. This information can be used to decide the capabilities of the VIP agents issuing or receiving transactions. Again, the extraction of this information can be specified by the users through a configuration file, similar to the one in Table I.

After extraction, all of the information is saved, including the corresponding RTL hierarchies. This information will be shown on a GUI, as shown in Fig. 3. There are two panels in this GUI. The left panel displays the design hierarchies. And the right panel displays the information related to a selected bus interface. As there can be multiple interfaces under the same hierarchy, there will be several Tab pages on the right panel, each interface per Tab. Inside a tab page, the capabilities of the selected interface are shown on the top section, while the signal mapping between the bus protocol and the RTL is shown on the bottom. Moreover, the information that cannot be extracted or is not aligned with RTL will be highlighted (see Section II.B and II.C for more details).



Feature	Allowed Values	Capability
Issuing Capability		
Version(APB3.0 has PREADY)	2/3	2
Read/Write Capability	R/W / ALL	ALL
Address Width(Bit)	8/16/32/64	32
Data Bus Width(Bit)	32/64/128	32
Address Range	0x0~0xffff_ffff / ALL	
DV_Setting		
Activation(Active/Passive)	A/P / ALL	

Bus Signal	RTL Net	Comment
Clock/Reset		
pclk	🟥	
presetn	🟥	
Control		
psel	psel	
penable	penable	
paddr	paddr	
pwrite	pwrite	
pwdata	pwdata	

Figure 3. An example of extracted information shown in GUI.

B. *Manual Effort As A Complement*

In order to be able to extract a complete interface from RTL, all of the signals within this interface should follow the same naming convention (i.e., with the same prefix and postfix). Unfortunately, this is not always true. According to our experience, about 80% of interface signals follow the same naming rule, and hence can be extracted from RTL. That is to say, 80% of manual effort in binding VIPs can be saved. On the other side of the coin, the remaining 20% interface signals have to be manually input by the users. For example, generally speaking, the clock and reset signals do not follow the same naming rule with other signals. As a result, the users have to manually provide the clock and reset signals in order to form a complete interface. As for the information that has been successfully extracted, there is no guarantee that it will be exactly correct. Therefore, it is vital that the users can review and revise the extracted information and to input those that cannot be extracted, before rushing to generate the verification environment. We have provided an easy-to-use GUI (as shown in Fig. 3) for the users to access the extracted information and to collect the users' input. Through the GUI, the users are able to trace an instance by its hierarchy, to review and revise the extracted information, and to input the information that is still missing or is incorrect.

C. *Correlation With RTL*

To ensure the obtained information is well aligned with RTL, this information is then checked against RTL. The checking results will be back-annotated to the GUI automatically. From this GUI, the users can know whether the information is correct. It could ensure the correctness of the verification environment generated from this design information, thereby relieving the users from tedious debugging work.

There is another benefit of this checking mechanism: it can help to make the acquired design information keeping up with RTL changes. Once the RTL is changed, the design hierarchy might be changed, and the bus interfaces might also be different from the original ones. In other words, the original information might be out of date. The users need to review the information again to ensure its correctness. However, it is painful and time-consuming to manually review the information one by one. In our implementation, the original information will be checked against the new RTL. Once there is an error found, the users will be prompted to make the corresponding revision.

D. *Correlation With Design Specifications*

Besides RTL, it is also possible to check between the acquired design information and some design specifications. For example, if we have a memory map (See Table II for example.) indicating the bus interfaces on corresponding instances, we are able to do cross-checking between the memory map and the extracted bus interfaces. Specifically, we can check whether the hierarchy information inside the memory map is correct, and whether the types of bus interfaces match those in the acquired design information, etc. Moreover, we can also check the addresses inside this memory map to see if there exists overlapping or an unexpected gap. In this way, we are able to achieve the consistency between RTL, design specifications and the acquired design information. This is helpful in building the verification environment.

E. *Information From Previous Designs*

Using the approach described above, the effort in building the verification environment is greatly reduced. However, it should be noted that the users' manual effort is still required. As will be explained below, the users' manual effort can be further reduced by exploiting the design information from previous designs.

In today's SoC designs, many of the modules and/or IPs are reused among different designs. In other words, the RTL codes of many modules remain unchanged or only contain minor changes when migrating from one design to another. As a result, the information that we have acquired from previous designs can be reused. The users will only need to run the checker described above, to check whether the design hierarchies have been changed, and to examine if any of the design information has been revised. If an instance is simply moved from one hierarchy to another, the users merely need to revise the hierarchy of this instance, instead of re-extracting and re-inputting the design information. If the RTL of this instance is not dramatically different from the original one, the users just need to revise the design information accordingly. Therefore, by reusing the design information acquired from previous designs, we are able to reduce users' manual effort to the minimum.

III. VERIFICATION ENVIRONMENT FOR SINGLE MODULE

For module-level verification, as the design is relatively simple, the above flow can be simplified into a push-button solution. Specifically, the testbench is immediately generated after the information extraction. The users can review and revise the generated testbench directly.

Take a bus bridge type design as an example. Based on the information extracted from RTL, we can know the bus protocol of each interface and configure the VIP accordingly. A typical verification environment involves DUT instantiation, VIP interface connections, VIP construction, stimulus and checking mechanism etc., as shown in Fig. 4. So the following templates based on UVM methodology will be generated. The users can do further customization based on these templates if necessary.

1. Testbench's top files for DUT instantiation, VIP interface connection, VIP agent instantiation, and VIP configuration.
2. Default scoreboard connections, i.e., TLM connections from VIP agents' subscribers to scoreboard components.
3. A text file for transaction constraint template. Using the method "Dynamic Constraint" [2], the text file can be parsed to generate transaction constraints. (See below for more details.)
4. Functional coverage definition based on bus capabilities.
5. A script to run simulations, e.g., a Makefile that includes default compilation and simulation command.

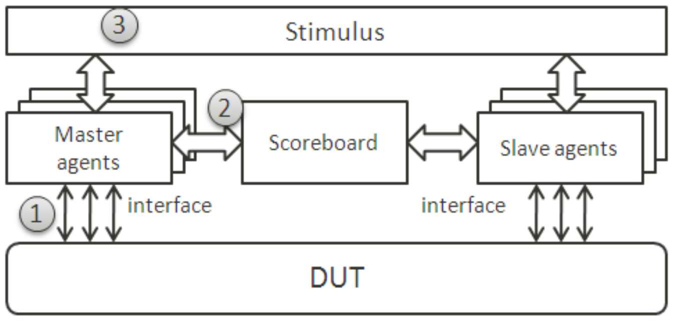


Figure 4. A typical UVM verification environment.

After the environment is built, the users will only need to do the following before running a simulation:

1. Adjust a few signal connections if necessary;
2. Add some VIP constraints;
3. Modify the default VIP configurations.

Then the users could simply type in "make" to run the testbench and get the simulation results.

Take the bus interfaces in Fig. 3 for example. The testbench's top file as shown in Fig. 5 will be generated. Here the interface connections "user_interface.svi" is shown in Fig. 6. Similarly, template codes for UVM agents, UVM envs, UVM test classes, etc. can be generated.

```
/// Testbench's top
module top;
  // Customized transaction classes per bus agent
  `include "user_transactions.sv"

  // UVM compilance environment,
  // including all necessary components
  // for bridge verification
  `include "user_env.sv"

  // Template test files.
  // The users can customize it if necessary.
  `include "user_test.sv"

  // DUT instantiation
  dut_top  u_dut();

  // VIP interface connections. See Fig. 5.
  `include "user_interface.svi"

  // UVM run
  initial begin
   run_test();
  end

endmodule
```

Figure 5. Testbench's top file for DUT "u_sramrom" in Fig.3.

```
// user_interface.svi

// Define a macro for RTL hierarchy.
`define APB_SLV_SRAMROM  \
        top.u_dut.u_infrasys.u_sramrom
apb_slv_if   intf_apb_slv_sramrom(
 .paddr (`AHB_SLV_SRAMROM.paddr),
 .psel   (`AHB_SLV_SRAMROM.psel),
 // all other APB signals
);

`define AXI_SLV_SRAMROM_A \
        top.u_dut.u_infrasys.u_sramrom
axi_slv_if   intf_axi_slv_sramrom_a(
 .araddr   (`AXI_SLV_SRAMROM_A.ARADDR),
 .awaddr (`AXI_SLV_SRAMROM_A.AWADDR),
 // all other AXI signals
);

// Connect to VIP agents through uvm_config_db
initial begin
 uvm_config_db #(virtual apb_slv_if)::set(
     null, "uvm_test_top.env.APB_SLV_SRAMROM",
     "vif", top.intf_apb_slv_sramrom);
 uvm_config_db #(virtual axi_slv_if)::set(
     null, "uvm_test_top.env.AXI_SLV_SRAMROM_A",
     "vif", top.intf_axi_slv_sramrom_a);
end
```

Figure 6. Interface connections for DUT.

For bus bridge type verification, we have a mature verification library [2] that contains stimulus and scoreboard template classes. For example, instead of generating the constraints directly, a text file as shown in Fig. 7 is generated based on the 'Capability' information in Fig. 3. In this text file, some frequently used transaction constraints are defined. The verification library will parse this text file during simulation, and randomize the transactions based on the selected master VIP and the target address region. This method is called Dynamic Constraint [2]. For example, in Fig. 7, if the AXI master "AXI_MST_A" wants to access REGION_X, it can only send READ transaction, and the burst size can only be 32 or 64. Using this method, the users can run simulations with limited verification knowledge. Even those who are not familiar with constraint random verification (CRV) can benefit from this method. Besides, from the point of the users, it is much easier to review the text file than to review the stimulus and scoreboard classes. Finally, the users can override the template classes if necessary.

```
AXI Protocol Constraint
Name             Direction    Burst_length    Burst_Size    Valid_ID
AXI_MST_A          ALL          ALL             32/64         ALL
AXI_MST_B          ALL          10~16           16/32/64      0~8
REGION_X           RD           ALL             ALL           ALL
REGION_Y           WR           ALL             32            ALL
```
Figure 7. A text file used for creating transaction constraints.

IV. VERIFICATION ENVIRONMENT FOR SUB-SYSTEM OR WHOLE CHIP

For sub-system or chip level verification, the structure of the verification environment is not much different from that of module-level verification. However, the design complexity increases dramatically, as there might be hundreds of bus interfaces. Therefore, we will need additional information that describes the overall picture of the bus fabrics. For example, in order to decode the addresses correctly, we will need a pre-defined memory map table as shown in Table II.

In this table, a memory region is defined with a unique name, along with its starting address and memory size. In addition, some essential information is also required. This information can be used to help extract design information from RTL. For example, the bus location and protocol type can be used to indentify bus interfaces in RTL. Every time a memory map is submitted, the cross-checking between it, the RTL code and the acquired design information will be performed. This can help to find errors in early stage, thereby reducing the effort to find bugs when a simulation fails.

Table II. MEMORY MAP EXAMPLE

Region Name	Start Address	Size	Protocol	Bus Location
SRAM	0x0000_0000	0x1000_0000	AXI	`TOP.u_sram
USB0	0x1000_0000	0x0001_0000	AHB	`TOP.u_usb0
USB1	0x1001_0000	0x0001_0000	AHB	`TOP.u_usb0
SPI	0x1002_0000	0x0001_0000	APB	`TOP.u_spi
DRAM_BANK1	0x8000_0000	0x1000_0000	AXI	`TOP.u_dram0
DRAM_BANK2	0x9000_0000	0x1000_0000	AXI	`TOP.u_dram1

With the design information (acquired as described in Section II) and this memory map table, the testbench of a sub-system or the whole chip can be generated, similar to the one in Section III. Again, for bus bridge type verification, the aforementioned verification library can be used to generate template classes for stimulus and scoreboard. Furthermore, the memory map in Table II can be turned into executable System-Verilog code to check the address decoding.

V. EXPERIMENTAL RESULT

We have applied this solution to verify bus fabrics in designs of different scales, i.e., module-level, sub-system-level and chip-level.

1. For module-level verification, the push-button solution described in Section III is adopted. With this solution, a typical testbench with less than 10 bus interfaces can pass simulation within 4 hours.
2. For a sub-system with 30 bus interfaces (5 different types of bus protocols in total), we are able to setup the verification environment and pass the first pattern within 4 days. Besides, coverage closure can be reached within 1.5 weeks. As the solution becomes more and more mature, it is estimated that the verification can be finished within 1 week.
3. For chip-level designs, this solution can also speed up the building of the verification environment dramatically, and improve the stability of the testbench codes in the meantime. Generally speaking, adopting this solution can reduce the overall time from months to weeks. The time spent in building the verification environment before and after deploying this solution is listed in Table III.
   a. *Design Information Collection*:
   Previously, we needed 3 weeks to collect the information regarding bus interfaces, VIP configuations and design limitations. Using this solution, we need less than 1 day to prepare the configuration files, and to extract the desgin information. After that, the design team can finish reviewing the information within 3 days. In total, the collection of design information can be finished within 1 week.

b. *Testbench Generation*:
Previously, we needed about 2 weeks to connect the VIP interfaces, to configure the VIPs, and to pass the first pattern. With the help of this solution, all of these codes can be generated by a simple script in 1 day. The users will only need to do a few refinements (as described in Section III) before running a simulation. The actual effort in writing codes is greatly reduced.

c. *Environment Stabilization*:
Previously, there were a lot of human errors in the hand-coded testbench. We needed 4 weeks to pass the regression test for the first pattern. After using this solution, the environment is always well aligned with the RTL. As most of the codes are generated automatically, the time spent in debugging is greatly reduced. Now the environment can be stabilized within 2 weeks.

Table III.
COMPARISON OF THE TIME SPENT IN BUILDING CHIP-LEVEL VERIFICATION ENVIRONMENT

	Previous	Now
Design Information Collection	3 weeks	< 1 week
Testbench Generation	2 weeks	1 day
Environment Stabilization	4 weeks	2 weeks
Total	9 weeks	3 weeks

From the perspective of project execution, less file revisions mean less effort in writing codes, and lead to faster iteration. So we will use file revision counts to estimate the users' effort. The file revision counts of 3 SoC designs (1 does not use this solution, and the other two use this solution) are shown in Fig. 8. The 3 designs all contain ~180 bus interfaces. After deploying this solution, the average revision count per file is reduced from 11.1 to 3.4 and 3.6 respectively. Therefore, it is concluded that the users' effort can be reduced by about 70%.

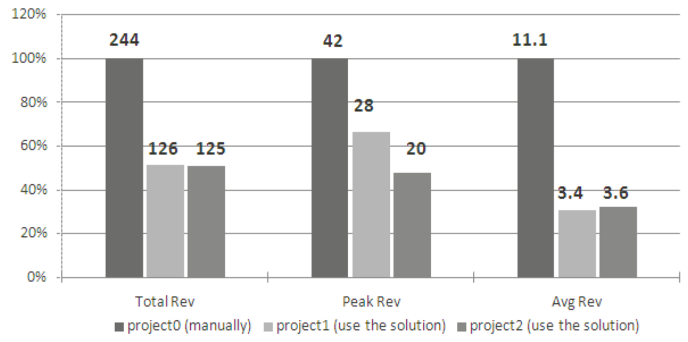


Figure 8. File revision counts before and after using the solution in this paper.

VI. SUMMARY

In this paper, we have proposed a systematic solution to extract design information from RTL, to provide an easy-to-use GUI for the users to review and input, to check the acquired design information against RTL, and to build the verification environment based on the information. This solution can be applied to designs of module, sub-system and chip level. In addition, for module-level verification, we have further simplified this solution into a push-button one. Using this solution, the time spent in building the verification environment can be greatly reduced. Moreover, the generated testbench is ensured to keep aligned with RTL changes. Finally, we have compared the execution of 3 different projects before and after deploying this solution. It is found that the users' effort can be reduced by about 70% after using this solution.

REFERENCES

[1] Avidan Efody, *Wiretap your SoC Why scattering Verification IPs throughout your design is a smart thing to do*, DVcon 2014.
[2] Yunyang Song, *Adopt Dynamic Binding to Isolate Stimulus Code Gracefully for Booming Complex SoC Bus,* DAC-DT 2014