Test bench Framework for Fully Automated Register Tests of Numerous IPs in SoC

Wonyeong So[1], Minje Kim[2], Jihye Lim[3], Sunil Roe[4], Youngsik Kim[5], Sunil Brain Choi[6]
Samsung Electronics Co. Ltd., Seoul, Korea
([1]wonyeong.so; [2]mjstyle.kim; [3]jihye420.lim; [4]sunil.roe; [5]ys31.kim; [6]seonilb.choi@samsung.com)

***Abstract*- The exponential increase in transistor density has escalated the design complexity of SoC(System on Chip), leading to longer verification times, especially as the number of integrated IP(Intellectual Property) rises. Verification engineers spend a significant amount of time creating and maintaining numerous IP access test benches. Verification engineers face the challenge of creating and maintaining numerous test benches, which is time consuming and increases their workload. Moreover, specifications that exist in various forms, such as natural language or specific formats, can be ambiguous and sometimes challenging for verification engineers to interpret. Additionally, verification holes may arise when verification engineers fail to recognize continuously updated specifications. The verification holes increase the likelihood of potential SoC bugs. This paper introduces a framework for automatically generating IP register test benches from a given specification. The framework aims to reduce the potential SoC bug occurrence rate, alleviate the workload of verification engineers, and improve verification efficiency through the test bench generation process. The ultimate goal is to shorten the verification TAT(Turn Around Time) in SoC projects.**

I. BACKGROUND

Over the past few decades, continuous advancements in process technology have led to an exponential increase in transistor density per chip. From a verification perspective, this trend indicates that the integration of numerous IPs into SoCs has heightened design complexity, thereby increasing the time required for verification. However, as the number of integrated IPs increases, the time required for verification can increase regardless of the design complexity. For example, as the number of IPs increases, the engineers need to create and maintain more test benches for SoC level register tests, such as read and write tests, for each IP. Maintaining a large number of test benches is relatively time consuming, which can impose a burden on engineers.

SoC verification starts with verification engineers interpreting and understanding the SoC specification. The specification can exist in various forms. For example, it could be in hardware description word format or an Excel format containing various connection details. It can also exist in the form of emails or chat messages, and even verbally. The verification engineers must understand and interpret these various specifications in order to create verification scenarios. If the verification engineer misinterprets the specification or if the specification itself is incorrect, there is a possibility that inaccurate verification scenarios may be created. SoC verification using inaccurate scenarios undermines the reliability of the verification and can lead to an increased frequency of bugs. For this reason, creating accurate verification scenarios is a critically important factor.

The challenge is that both the specification and verification scenarios are generated by humans. This implies that there is always a possibility of issues arising due to human error. For example, issues may arise if the specification is insufficiently detailed, or if the verification engineer is unaware of updates or changes made to the specification. These challenges are among the significant hurdles in modern SoC verification[1].

While methods such as regular reviews of specifications and verification scenarios can help address these issues, they are not a fundamental solution. This is because these reviews are also conducted by humans, and thus still subject to human error. Moreover, reviews can be time-consuming, and in some cases, the test bench may need to be updated to align with the modified specification, which can further add to the time required. These are inefficiencies that arise due to human error. Test bench modifications, in particular, can occur frequently due to the frequent specification changes common in the early stages of a SoC project. This can be considered unnecessary work and contributes to increased fatigue for verification engineers.

Moreover, generating all test benches from the specification can be challenging. If the specification is ambiguous or written in natural language, making it difficult to parse, test bench generation may not be feasible. Additionally, parsing the specification to generate a small number of test benches may not yield results that justify the time invested. Consequently, an ideal case for this approach would require the specification to follow a consistent format and allow for the generation of a large number of test benches. In this paper, register access test was selected as a suitable case. This is because register tests have a predefined list of IPs to be tested, and the number of test benches required is significantly large.

Register access tests for IPs are the most fundamental and essential tests during SoC verification. These tests are relatively straightforward in terms of scenario complexity. This is because they mostly involve simply attempting

register read and write operations. However, performing these tests is essential and highly important during SoC verification. This is because conducting these tests ensures accessibility to the internal registers of the IP, which is necessary before performing more complex tests. However, managing the test benches for this can be time-consuming. As mentioned earlier, a test bench is needed for each IP, and frequent specification changes lead to frequent modifications of the test benches as well. Moreover, such test benches are not well-suited for reuse across SoC projects, as the specifications for IPs can vary significantly between projects.

If the register access test bench can be automatically generated from the given specification, it will significantly reduce the verification engineer's workload while also shortening the verification TAT. This paper introduces a framework for generating a register access test bench using the given specification. This framework aims to drastically reduce the time required for register access testing.

II. Proposed Method

The test bench framework consists of five components, including Specification Collector, Test Bench Generator, Test executor, Debugger, and Reporter. Each component is based on Python scripts, while the reporter was developed using GO language. The reason the reporter was developed in GO is that handling a large volume of test results requires a fast response time, and GO was deemed more suitable for this purpose than Python.

A. Specification Collector

The specification collector gathers the necessary specifications for generating a register access test bench. The primary specifications to be collected are the lists of IPs to be tested. To collect the IP list, the collector parses the IP-XACT XML format files used for SoC integration, the SoC address map specification, and the UVM(Universal Verification Methodology) register model specification. The reason for parsing these three types of specifications is to detect potential errors in the specifications, which would be impossible if only a single specification were analyzed. Additionally, by exploring multiple specifications, the integrity of each specification can be guaranteed, and cross-checking can be performed.

The reason the collector parses the three specifications mentioned earlier is as follows. First, from the XML file, the names of the IPs integrated into the blocks of the SoC can be obtained. It is also a file format that is advantageous for Python to parse. Next, from the SoC address map specification, the collector can obtain the list of IPs included in each block of the SoC and the base address information assigned to each IP. The address map specification file is typically in Excel format, which is also convenient for Python to parse. Finally, from the UVM register model, the collector can obtain the list of IPs within the SoC, the UVM class handler names and base address information for each IP. The SoC verification process mainly takes place in a UVM test bench environment. Therefore, to conduct a register test, the UVM register class handler corresponding to each IP must be matched. For this reason, the collector explores the UVM register model.

After exploring the three specifications, the information obtained from each specification is compared. First, the obtained IP lists are compared to check for any mismatched IPs between the three lists. If any mismatched IP names are found, it is considered a specification bug, and an error log is generated to allow the verification engineer to identify it. The same comparison is made for the base address information of each IP, and if any mismatches are found, an error log is generated.

In addition to the basic read and write tests during the register access test, tests checking the security attributes can also be performed. The security test verifies whether the RTL implementation aligns with the security specification by checking access permissions based on different security levels between the master and the IP. For example, if an IP is set to secure mode, a non-secure master should not be able to access it. The security specification is also managed in a structured file format, and in the case of the SoC, a TZPC(TrustZone Protection Controller) table can be an example of this. The collector also explores the security specification and collects the security specification information for each IP.

Once the specification collection is complete, the collector generates a YAML format file, which contains the collected specification information for each IP. This YAML format file is used later during the test bench generation process.

Figure 1 provides an example of the operational structure of the specification collector for IP_A and the generated YAML file. The generated YAML formatted file is used as an input file for test bench generation.

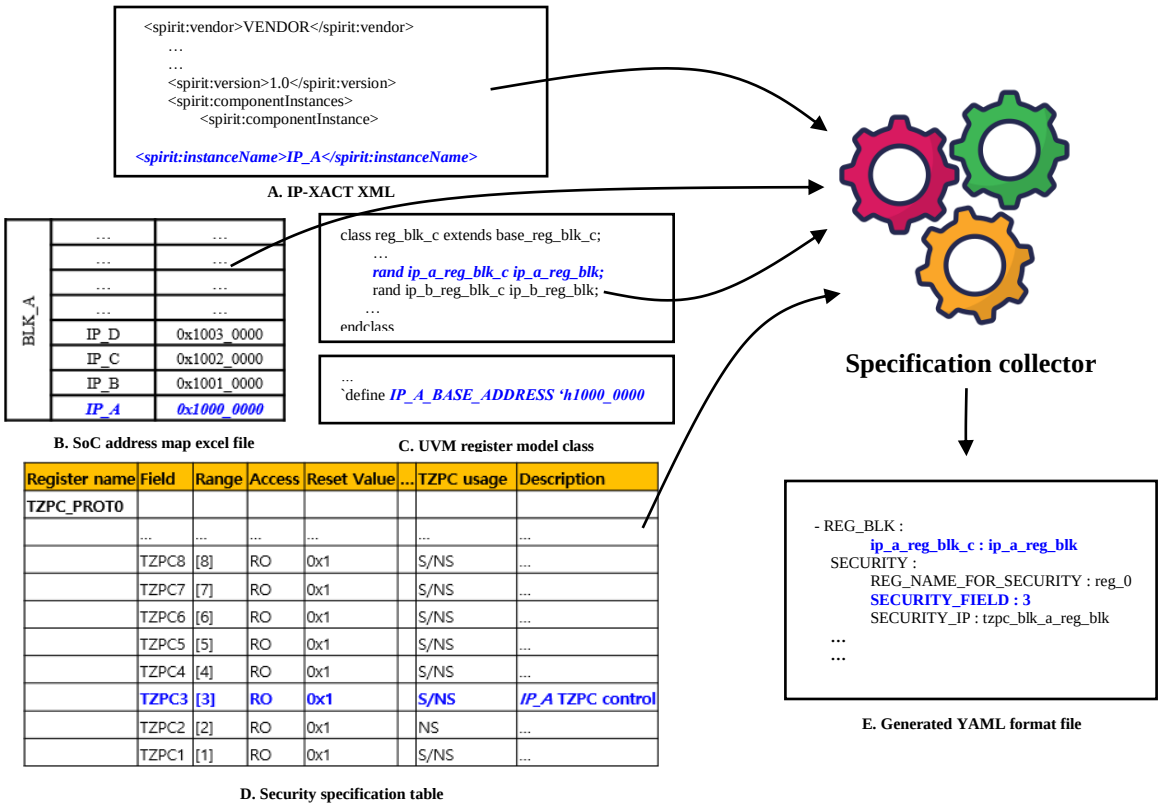

Figure 1. An example of the operation of the specification collector for IP_A

B. Test bench Generator

The YAML formatted file contains all the necessary information for creating register access test benches for all IPs within the SoC. The test bench generator uses the given YAML data as input to generate register access test benches for each IP. The Python Jinja2[2] library is used in this process. The Jinja2 library leverages templates to render and is well-suited for generating a large volume of test benches efficiently. However, it requires that templates be predefined, which may make it less suitable for test benches that undergo frequent changes. However, the structure of a register access test bench is simple, consisting mainly of repeated read and write operations, making the test sequence straightforward. As a result, changes to the test bench are minimal. Therefore, it is suitable for Jinja2 template-based rendering, as changes to the templates can be minimized.

Table 1 shows the Jinja2 template used to generate register access test benches for individual IPs. To facilitate efficient template management, template macros were utilized to establish the overall template structure. Table 2 shows the Jinja2 template macros. Template macros are a set of variables that are updated for each IP.

Table 1. The template for the generation of individual IP register test benches

```
1: {%- import "template_macro" as register_macro %}
2: {%- call(register_info) register_macro.REGISTER_LOOP(IP, "", 0) -%}
3: class {{register_info.register_name}}_register_rw_access_vseq_c extends base_vseq_c;
4:     register_access_with_security_seq_c register_access_with_security_seq;
5:     …
6:     virtual task body();
7:         register_access_with_security_seq = register_access_with_security_seq_c::type_id::create("register_access_with_security_seq");
8:         register_access_with_security_seq.reg_block = p_sequencer.p_reg_blk.{{register_info.register_inst}};
9:         …
10:        {%- if register_info.security is defined %}
11:        {{- register_access_with_security_seq(register_info.security_reg_block,"security_block","\"%s\""|format(register_info.security_reg_block)}}
12:        {{- register_access_with_security_seq(register_info.security_field,"security_field",register_info.security_field)}}
13:        …
14:        endtask:body
15:endclass: {{register_info.register_name}}_register_rw_access_vseq_c
```

Table 2. The template macro

```
1:{%- macro REGISTER_LOOP(REG_COMPONENT, TAG="sfrgen", indent=0, comment='//') %}
2:    {%- for register_class in REG_COMPONENT.REG_BLK %}
3:    {%- set register_inst = REG_COMPONENT.REG_BLK[register_class] %}
4:    {%- set register_name = register_inst|replace("_reg_blk","") %}
5:    {%- if REG_COMPONENT.SECURITY is defined and REG_COMPONENT.SECURITY != None %}
6:        {%- set security = 1 %}
7:        {%- set security_reg_block = REG_COMPONENT.SECURITY.SECURITY_BLOCK %}
8:        {%- set security_reg_num = REG_COMPONENT.SECURITY.SECURITY_REG_NUM %}
9:        {%- set security_field = REG_COMPONENT.SECURITY.SECURITY_FIELD %}
10:         …
11:       {%- set REGISTER_ACCESS_INFO = {
12:           'register_class' : register_class,
13:           'register_inst' : register_inst,
14:           'register_name' : register_name,
15:           'security' : security,
16:           'security_reg_block' : security_reg_block,
17:           'security_field' : security_field,
18:           …
19:       } %}
20:   {{- caller(REGISTER_ACCESS_INFO)}}
21:{%- endmacro %}
```

C. Test Executor

Once the test bench has been successfully generated, the next step is to use the generated test bench to perform an actual IPs register access test. While verification engineers can perform the tests manually, the sheer volume of test benches often exceeding hundreds makes this approach highly inefficient. To resolve this, the framework extends its functionality to include test execution after generating the test benches. The test executor begins with DUT(Design Under Test) compilation and performs regression testing for the register tests automatically.

D. Debugger

Errors may occur during register access tests. Errors can arise from various causes, and verification engineers must debug them to ensure the tests pass. For example, a master might receive an unexpected error response while reading from or writing to a register of a specific IP. Other issues such as a bus hang or a mismatch between the write data and the read data could also occur. These errors may require varying amounts of debugging time depending on the skill level of the verification engineer. Sometimes they are resolved quickly, but at other times, they can consume a significant amount of time for debugging. Moreover, the increasing complexity of SoC architectures tends to further extend the time required for debugging. As a result, the more the debugging time is reduced, the easier it becomes to shorten the verification TAT. The test bench framework aims to reduce debugging time by building a basic level internal debugger and utilizing a BTS(Bus Trace System) based debugger[3].

In a UVM-based verification environment, errors are reported as UVM_ERROR. Additionally, verification engineers can include custom messages when printing UVM_ERROR to the simulation log. This means that the test bench can attach tags to UVM_ERROR and print them in the simulation log, enabling the identification of errors using these tags. The debugger includes solutions for each of these error tags. If an error with a specific tag is printed in the log, a corresponding solution is provided. The solutions are predefined in the test bench for these errors, which are predictable in nature.

However, unexpected errors may occur during simulation. Design related bugs can lead to bus hangs or unexpected error responses. Since these types of errors are not predefined, solutions cannot be provided for them. To easily debug such errors, the framework utilizes BTS. BTS can pinpoint the exact point of problem such as bus hangs and error responses. BTS is implemented as a UVM monitor within the test bench and is activated through the VIP(Verification IP) callback mechanism. If a timeout error callback occurs, it is identified as a bus hang, triggering the corresponding BTS debugger. Similarly, if an end response callback indicates an error response, the relevant debugger is invoked for debugging.

Once debugging is complete, the issue points are printed in the simulation log. As a result, the verification engineer can accurately identify the issue points related to bus hangs and error responses. Figure 2 illustrates the overall operation of the framework debugger for errors that occur in the simulation log. Three types of error logs are displayed, including predefined errors with error solutions, error responses, and bus hang error logs. In the case of error responses and bus hangs, the debugging results from BTS are also included.

[0] **UVM_ERROR** (reg_access_with_security_seq_c) **[SFRSEC][NOSECUREG] security_reg_num_for_rd is not given.**
[0] UVM_INFO (reg_access_with_security_seq_c) **Debugging guide** → **Error tag**
[0] UVM_INFO (reg_access_with_security_seq_c) **1. Open sim.log file in log path**
[0] UVM_INFO (reg_access_with_security_seq_c) **2. Check the "[SFRSEC][NOSECUREG]" issue in log file**
[0] UVM_INFO (reg_access_with_security_seq_c) **3. Assign the correct value to the security_reg_num_for_wr/rd variable. Below is an example code.**
[0] UVM_INFO (reg_access_with_security_seq_c) **sfr_access_with_tzpc_seq.security_block = "security_ip_a_reg_blk";**
[0] UVM_INFO (reg_access_with_security_seq_c) **sfr_access_with_tzpc_seq.security_reg_num_for_rd = 0;** → **Error solution**
[0] UVM_INFO (reg_access_with_security_seq_c) **sfr_access_with_tzpc_seq.security_reg_num_for_wr = 2;**

A. Predefined error log and error solution

…
[5701957] UVM_INFO (**bts_c**) **BTS::Error response debugger is called**
bts_intf_inst:: flush function is called → **Debugged by BTS**
[5701957] UVM_INFO (**bts_c**) BTS::Error response occurrence instance : ***top.dut.BLK_A.AXI2APB***
[5701957] UVM_INFO (**bts_c**) BTS::Please refer bts directory to get more detail about debug result
[5701957] **UVM_ERROR** (AXI::cdn_ps_amba_if::prd) [**SLVERR**] Error response occured. UVM_READ, addr = 0x1000_0100, data = 0x0000_0000, id = 0x22 (ip_a_reg_blk.reg_a)
…

B. Error response log and debug result by BTS

[23971539] UVM_WARNING (CDN_AXI_NONFATAL_WARNING_VR_AXI1001_WRITE_BURST_RESPONSE_TIMEOUT_AND_DISCARDED)
Write burst response timeout
…
Timeout burst: kind:INCR address: address:0x0012000040 id:37 len:1 size:WORD access:NORMAL
Title: Write burst response timeout
…
[23971539] UVM_WARNING (DENALI_CDN_AXI_CB_Error) Agent master dropped transaction with transaction id: -1
[23971539] UVM_INFO (**bts_c**) **BTS::Timeout debugger is called** → **Debugged by BTS**
[23971539] UVM_INFO (**bts_c**) BTS::Bus hang occurrence instance : ***top.dut.BLK_B.IP_B***
[23971539] UVM_INFO (**bts_c**) BTS::Please refer bts directory to get more detail about debug result

C. Bus hang log and debug result by BTS

Figure 2. Overall operation of the framework debugger for errors that occur in the simulation log

E. Reporter

Even if thousands of simulations end successfully without errors, it would be quite difficult if there is no way to check this. Additionally, checking the results of each simulation one by one is very inefficient. In other words, a platform that allows for a quick overview of simulation results is essential. Through the reporter in the test bench framework, verification engineers can view the results of simulations at a glance. The reporter was developed in Go, as it needs to handle large amounts of simulation result data and deliver high responsiveness, which is relatively faster than Python. EDA(Electronics Design Automation) vendors provide tools for reviewing simulation results. However, these tools appear to be over-specified for register access tests, so the reporter was developed.

The reporter provides three types of views. The first view displays the register test results for each block within the SoC, including the pass and fail status and pass rate for each block. Figure 3 shows the first view that verification engineers see when they call the reporter. This view displays the entire regression suite list along with the test status for each entry.

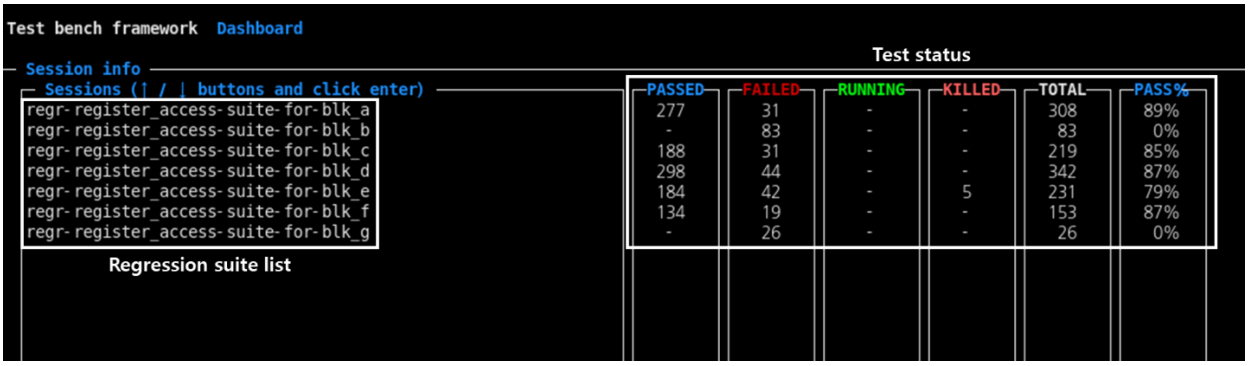

Figure 3. The first view of reporter

The engineer can select a regression suite list to view the register access simulation results for a specific block. Figure 4 illustrates the results of the register access test regression for block F. This view provides an overview of the test results for the IPs included in each block at a glance. Additionally, if the test result for a specific IP fails, it is possible to identify the type of error that occurred.

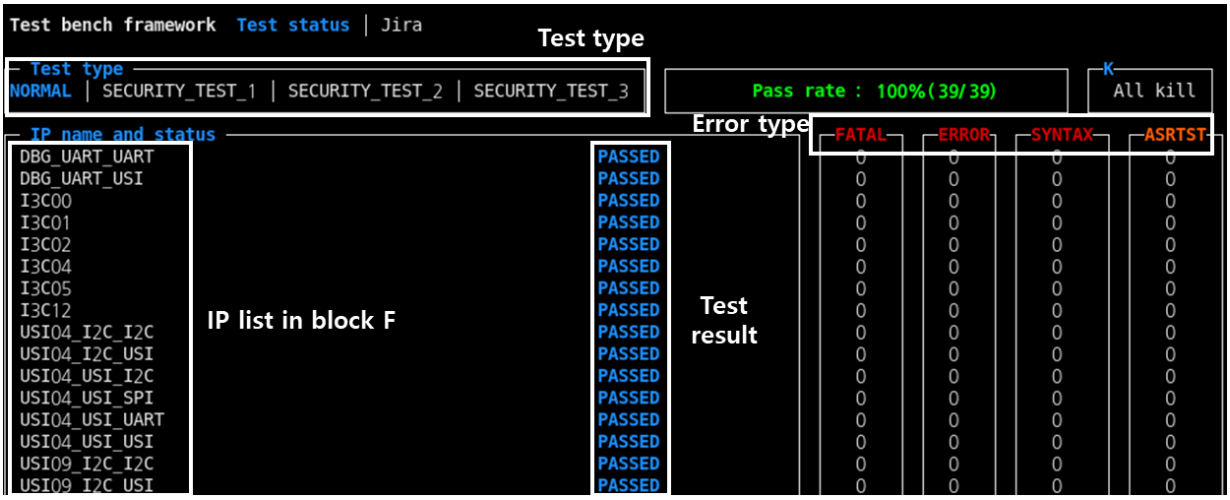

Figure 4. Register access test regression result of block F

In SoC projects, the RTL corresponding to the DUT is typically managed using version control tools such as Git or Perforce. Additionally, the RTL is released to the verification team at regular intervals with a specific label. In principle, simulations must be performed for all testable scenarios each time the RTL is released. However, this is often impractical due to the limited resources of verification engineers and the relatively short timeline of SoC projects. If a test bench tailored to the released RTL label is automatically generated and simulations are executed accordingly, this goal could be partially realized.

The test bench framework is automatically triggered by using the RTL label release as an event. In this paper, the framework is triggered using the ‘p4 trigger’ command, one of the Perforce commands, as an event. If the RTL is managed with Git, the framework can also be triggered using Git Actions.

The p4 triggers is a command that defines and executes scripts when specific actions are performed in a designated directory[4]. Through this, the test bench framework is automatically executed each time an RTL label file is registered in Perforce. However, there are several prerequisites for automatically executing the framework through ‘p4 trigger’. First, the RTL label files must be periodically created in predefined, agreed-upon paths. This is because the command requires specifying a directory for the event trigger. If the path is too broad, the framework may be executed unintentionally. Additionally, the RTL label files must follow a predefined naming pattern. This is also to prevent unintended triggering of the framework.

When an RTL label file is committed to the specified path, the framework operates to generate an IP register access test bench and creates a Perforce change list for the related test bench files. This process is repeated each time an RTL label is committed during the SoC project timeline.

As a result, for each RTL label, the IP register access test bench is automatically generated to match the label, and the simulation is also performed automatically. In other words, the framework implements full automation for IP register access tests in SoC projects, from test bench generation and simulation execution to debugging, without the need for intervention from verification engineers. Figure 5 illustrates the overall working flow of the test bench framework.

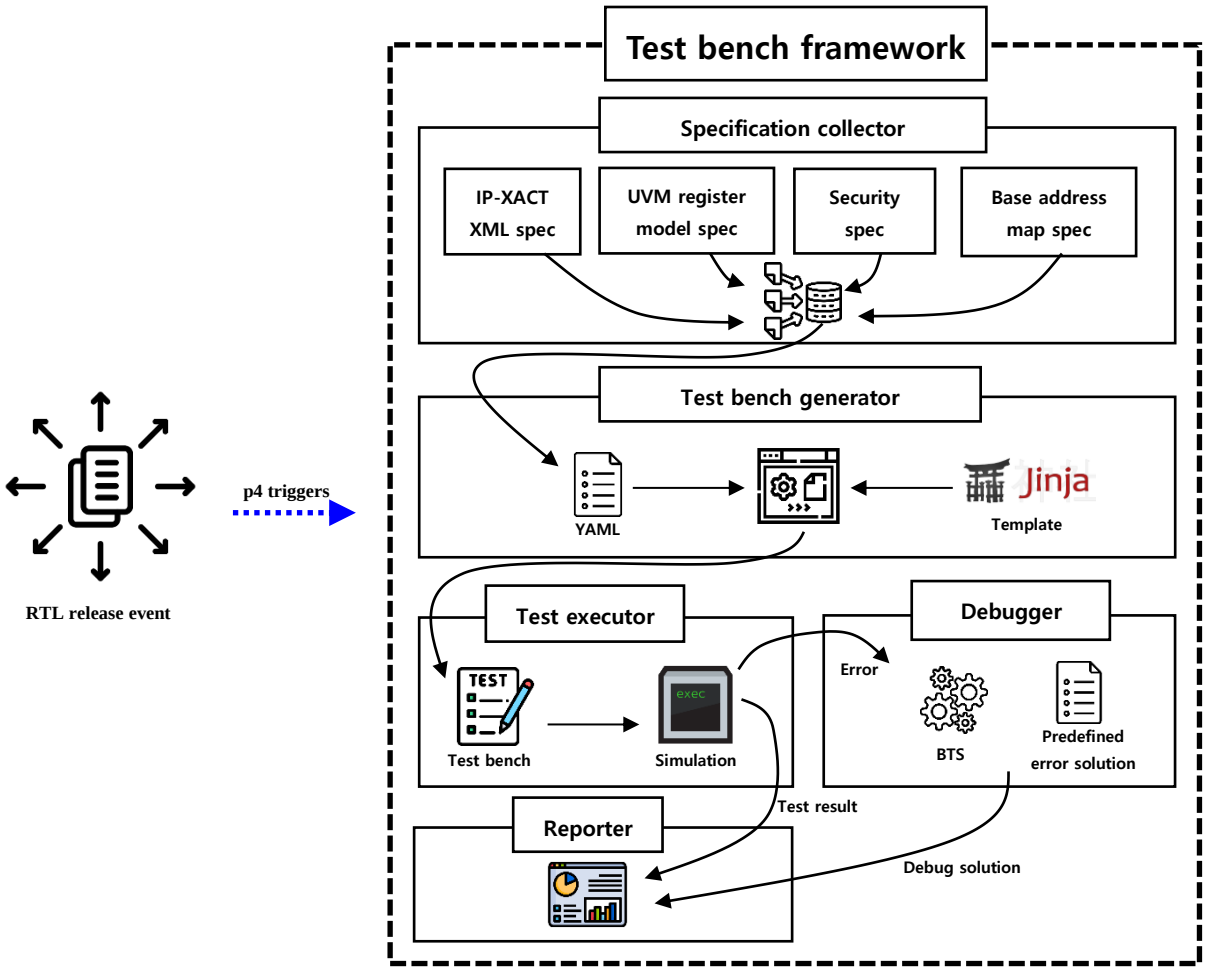

Figure 5. Working flow of the test bench framework

III. RESULT

Table 4 shows the total number of IPs per project, the number of IP register access tests related to them, and the total number of tests performed per project in the five recent SoC projects conducted by our company. Although there is variability in the total number of IPs and tests across projects, it can be observed that, on average, approximately 35% of the total tests are related to IP register access. The reason the number of tests is significantly higher than the number of IPs is that, in addition to basic register access tests, there are also security related tests. Furthermore, the number of IPs in each SoC project matches the number of IP register access test benches.

Table 4. The proportion of IP register tests relative to the total number of tests

SoC project	IP(#)	IP register tests(#)	Total tests(#)	Ratio
A	1,845	5,418	13,894	**0.39**
B	1,632	4,804	13,687	**0.351**
C	1,279	3,798	10,501	**0.3617**
D	1,184	2,839	8,570	**0.3313**
E	1,756	6,595	18,127	**0.3638**

As a result, the test bench framework can cover approximately 35% of the test benches per project on average, and it also eliminates the time required for maintaining these test benches. Additionally, the register access tests can be automatically executed for each RTL label, which is periodically released for every project. This is significant in that it can proactively prevent verification holes for IPs with specification changes that the verification engineer may not be aware of. Furthermore, since the tests are executed for each label, the framework can also serve as a sanity check for IP register access.

IV. CONCLUSION

Through the IP register access test bench framework, approximately 35% of the tests related to each SoC project can be automatically generated with their corresponding test benches. In addition to generating test benches, the process can also automate simulation runs and debugging in the event of errors. This is significant in that it reduces the time spent on performing IP register access tests and maintaining test benches, thereby efficiently shortening the verification TAT. This enables effective reduction of verification time, which is crucial as SoC architectures become increasingly complex, leading to longer simulation times in line with modern SoC development trends.

However, the true significance lies in the fact that the framework automatically performs sanity tests for IP specification changes that verification engineers may not be aware of, allowing for early detection of IP bugs. This allows for early improvement of verification quality and enables more efficient time allocation for verification engineers.

REFERENCE

[1] W. Chen, S. Ray, J. Bhadra, M. Abadir and L. -C. Wang, "Challenges and Trends in Modern SoC Design Verification," in IEEE Design & Test, vol. 34, no. 5, pp. 7-22, Oct. 2017, doi: 10.1109/MDAT.2017.2735383.

keywords: {IP networks;System-on-chip;Field programmable gate arrays;Formal verification;Observability;Computer architecture;Verification and validation;Emulation;Postsilicon validation;System-level validation;Hardware/Software covalidation},

[2] https://jinja.palletsprojects.com/en/stable/

[3] Wonyeong SO, "Bus Trace System: Automating Bus Traffic Debugging in IP-XACT Based SoC Beyond Traditional Debugging Methods" Mar 2024, DVCON U.S. 2024

[4] https://help.perforce.com/helix-core/server-apps/cmdref/current/Content/CmdRef/p4_triggers.html#p4_triggers