2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Overcoming SystemVerilog Assertions limitations through temporal decoupling and automation

Mattia De Pascalis, Xia Wu, Matteo Vottero, Jacob Sander Andersen

SyoSil ApS, Høje Taastrup, Denmark

(mattia@syosil.com), (xia@syosil.com), (matteo@syosil.com), (jacob@syosil.com)

***Abstract*—In this paper, the outcome of developing a reusable formal verification framework is presented. SystemVerilog Assertions (SVA) language is used to describe formal assertions, while the entire process has been automated through a framework coded in Python. Our method shows a possible approach to overcome two SVA's limitations: the lack of support to represent dynamic cycle delay (run-time re-configurable delay) and the absence of tasks/utilities that could automatically derive an invariant assertion. The solution decouples dynamic and static components of a delay: the first one is modelled with helper logic and it is moved out from the SVA logic, whereas the constant part of the delay is then used to write both direct and invariant assertions in SVA. Thus, dynamic cycle delays with and/or without constant uncertainties can be produced by the tool. The generated code has also been successfully ported in simulation with due arrangements to handle level-sensitive sequence controls. This methodology adoption not only reduced the learning curve for less experienced verification engineers, but it also secured a consistent architecture for solving this kind of problems.**

I. INTRODUCTION

Today, assertion-based verification (ABV) has been used as a crucial part of the modern functional verification methods. Low level assertions, usually implemented as white-box assertions, describe the exact behavior of transactions or pin wiggling. They are either bound directly on RTL module or embedded inside the simulation-based test bench. High-level assertions, which translate the requirements directly into machine language, can be used as the building elements in the formal based test bench. SystemVerilog assertion is in its nature a concise description of certain temporal logic. Therefore it has always been considered as the most efficient to verify timing accurate functionalities, and often as a complement to the transaction level modelling of the system.

A big portion of the assertions can be categorized as to measure the timing relations between several events or sequences. This gives us the idea to abstract and find a generic solution to cover most requirements in this category, and then to automate the process to make it consistent and scalable in the deployment. This is the fundamental motivation of this work.

Timing relations between two events can be as simple as "event A happened 5 clock cycles after event B", but often the requirements contain more variable or uncertain timing relations. We also wish to include timing checks between sequences too. And due to the temporal nature of the sequence, this adds one more level of complexity. One limitation of the SVA language is that the time range has to be specified and solved at compile time. We want our tool to be able to handle not only the use cases with static and fixed timing requirements, but also the more dynamic, run-time configurable delays and timing relations.

Furthermore, our goal is also to check the completeness of logical relations, i.e. both sufficiency and necessity of conditions should be checked. Depending on the use case, we generate both the direct assertion to check the sufficiency, and the inverted assertions (referred to as *invariant assertions* in the paper) to check the necessity. It is also possible to change the assertion property to cover property for the necessity check, which shows if the timing checks has been triggered at all.

The outcome of our work is a tool called *SVAGen*, which takes the input template in Python and automatically generate a set of assertions as an output. These assertions can be used in a formal based verification environment which exhaustively verify the timing relations for various events and sequences. The converging time of these assertions are tested and presented. Also the assertions can be used directly in a traditional simulation based test bench as part of the functional verification process. The coverage result can be generated and merged by EDA coverage tools. With little learning time, verification engineers with no SVA experience can produce high quality, customized assertions which verify different timing requirements of a module. The tool reduces the general verification time, improve verification quality, and leverage the organisation's SVA competence.

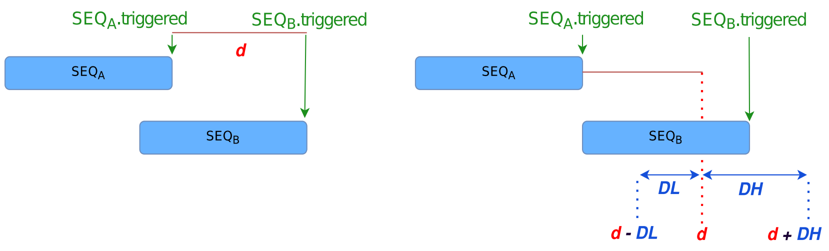


Figure 1. A fixed timing check scenario vs. a dynamic timing check scenario

II. Problem Definition

Our first step is to generalize the target use cases of our tool into an abstract model, so we can come up with a generic solution. We consider the following scenario as our starting point:

- Sequence A happened, then after a fixed delay of d clock cycles, sequence B happened.

This is depicted in the left part of Figure 1, and can be easily solved by writing 1-2 assertions. Now we add some complexity to the scenario to cover more scenario:

- Sequence A happened, then after a variable delay of d clock cycles, sequence B happened.

For certain use cases, it is still not enough. We want to specify even more uncertainty on top of the variable delay of d clock cycles, as depicted in the right part of Figure 1:

- Sequence A happened, then after a variable delay of d clock cycles, with some uncertainty of ±N cycles, sequence B happened.

This generalization covered most of the use cases in a certain customer project, therefore we take it as the target for our *SVAGen* tool. One of the major challenges we encountered when implementing a generic timing check, is to describe in concise SVA language the condition and the time range for one or the other sequence to happen. Translating the generalized target into pseudo SVA language, we get

$$SEQ_A.triggered \ |\!-\!> \ \#\#[d - DL : d + DH] \ SEQ_B.triggered \qquad (1)$$

$$SEQ_B.triggered \ |\!-\!> \ \$past(SEQ_A.triggered, [d - DL : d + DH]) \qquad (2)$$

The window `[d−DL:d+DH]` represents the valid temporal interval which *SEQ*$_B$ must trigger in: *d* is a dynamic delay, *DL(DeltaLow)* and *DH(DeltaHigh)*are static values. Method `triggered` is a SVA standard function, which tests whether its operand sequence has reached its end point at that particular point in time. Property (1) affirms that *SEQ*$_B$ must trigger within a range of *[d–DL:d+DH]* clock cycles after *SEQ*$_A$. Property (2) ensures that if *SEQ*$_B$ is triggered then *SEQ*$_A$ was triggered in a range of *[d–DL:d+DH]* clock cycles in advance.

To the best of authors' knowledge, SVA language constructs does not support checking re-configurable delays at run-time employing. Writing these two properties as they are and we will end up with compile errors. We must find other ways to handle the check of this kind of dynamic timing relations.

The limitations taken into consideration in our work are summarized in the following observation points (*OP*):

- *OP-1*. SVA's cycle delay operator shall only be used with constant values. Property (1) can be modelled with SVA only if *DL, DH* and *d* are constant values.
- *OP-2*. We must provide both the direct and inverted assertion to check the sufficient and necessity conditions. There is no standard SVA construct to automatically express (2).

In our case, (1) cannot be directly modelled in SVA because *d* can vary at run-time. And the same applies for (2) since *$past()* operator requires a static number of clock cycles.

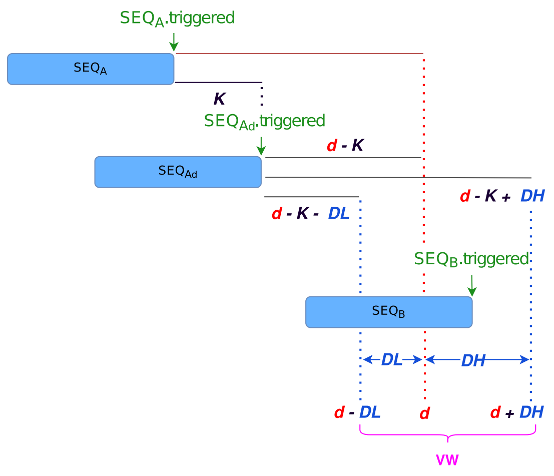


Figure 2. Use of a delayed version of SEQ$_A$ to eliminate the dynamic delay from the Validity Window (VW).

III. System Overview

A. Theoretic Solution and Implementation

As seen before, (1) describes a dynamic temporal relation between two sequences SEQ_A and SEQ_B. To solve our problem, we created a delayed version of SEQ_A, such that this new sequence is triggered K clock cycles after SEQ_A. Figure 2 shows the timing relationship between SEQ_A, SEQ_{Ad} and SEQ_B. Our reference is now shifted from SEQ_A to SEQ_{Ad}, so it is possible to re-write (1) to (3) and (4)

$$SEQ_{Ad}.triggered \mid\!-\!> \#\#[d - K - DL: d - K + DH]\; SEQ_B.triggered \tag{3}$$

Now we want this delayed sequence SEQ_{Ad} to happen at the earliest point of time where SEQ_B can happen. That is when the $K = d - DL$. And we can rewrite the rest of the check as in (4). As shown here, after we delayed SEQ_A by d-DL cycles, the time between SEQ_{Ad} and SEQ_B becomes a fixed number. And this will no longer be a problem to describe the check in SVA.

$$SEQ_{Ad}.triggered \mid\!-\!> \#\#[0: DL + DH]\; SEQ_B.triggered \tag{4}$$

To summarize, the solution is obtained by decoupling the dynamic and the static timing relations in this scenario. We first model the dynamic timing relations by creating a delayed version of the original sequence, which is delayed by d–DL clock cycles. Once we have removed the dynamic part from the equation, we are left with a static Validity Window (VW) with a size of $DL + DH$, which can be checked by simple assertions.

There are many ways to implement the two checks. One solution is to use a counter structure to book-keep the delay number. Then we use the SVA to check the if SEQ_B, happens inside the VW. Another way is to introduce two FIFO structure. One FIFO should track the dynamic delay of the SEQ_A, while the second represent the VW, with its length as DL+DH. The triggering of SEQ_A is stored in the first shift registers for d-DL cycles, and then being popped into the second. As soon as the recorded event enters second FIFO which represents VW, SEQ_B is expected to trigger within a range of *[0: DL + DH]* clock cycles.

In the FIFO implementation, we can derive the invariant property as: If SEQ_B is triggered, then an event is expected to be recorded inside the window. VW's emptiness translates into the assertion failure for this property, meaning that no SEQ_A was previously triggered. Consequently, the concept expressed by the inapplicable (1) and (2) can now be represented in SVA respectively by (5) and (6):

$$VW[DL + DH] \mid\!-\!> \#\#[0: DL + DH]\; SEQ_B.triggered \tag{5}$$

$$SEQ_B.triggered \mid\!-\!> VW \mathrel{!=} {}'0 \tag{6}$$

```
always @(posedge clk)
begin
  if(top.rst)
  begin
    ...
  end else
  begin
    A_s1_triggered <= S_ROSE(top.in_0).triggered;
    A_s2_triggered <= S_ROSE(top.out_0).triggered;

    A_window[`A_WINDOW_L - 1 : 0] <=  A_window[`A_WINDOW_L : 1];
    if(32'(A_delay - `A_DL)  > 2) A_window[`A_WINDOW_L] <= A_fifo[0];
    if(32'(A_delay - `A_DL) == 0) A_window[`A_WINDOW_L] <= S_ROSE(top.in_0).triggered;
    if(32'(A_delay - `A_DL) == 1) A_window[`A_WINDOW_L] <= A_s1_triggered;
    if(32'(A_delay - `A_DL) == 2) A_window[`A_WINDOW_L] <= A_s1_triggered_d0;

    A_fifo <= A_fifo >> 1;
    A_fifo  [32'(A_delay - 1 - `A_DL)] <= S_ROSE(top.in_0).triggered;
    A_s1_triggered_d0 <= A_s1_triggered;
    A_s1_triggered_d1 <= A_s1_triggered_d0;
    A_s1_triggered_d2 <= A_s1_triggered_d1;
  end
end

property p_A;
  A_window[`A_WINDOW_L] |-> A_s2_triggered;
endproperty: p_A

property p_A_inv;
  A_s2_triggered |-> A_window[`A_WINDOW_L];
endproperty: p_A_inv

a_A     : assert property(`CLK_SEQ p_A());
a_A_inv : assert property(`CLK_SEQ p_A_inv());
```

Figure 3. Implementation of the solution in SVA

When we were implementing our solution in real SVA code, we started out from a specific use case, shown in Figure 3. We intend to check the timing between two sequences: *S_ROSE(top.in_0)* and *S_ROSE(top.out_0)*. *S_ROSE* is a user defined sequence. The triggered event of these two sequences is first recorded into the A_fifo, which keeps track of the dynamic delay part. After the event has been shifted out of the A_fifo, it enters A_window, which tracks the VW part. A few corner cases have been handled explicitly, as shown in the code. The code has been written in a way that is later generalized as a template to generate assertions for similar use cases. The implementation provides us a generic solution to overcome the above-mentioned limitations *OP-1* and *OP-2*.

B. *Automation*

After we have implemented our solution, we reached the second phase whose goal is to abstract the SVA writing process. We wish to implement a push-button tool to generate consistent and scalable SVA code which can handle timing checks in this category. Our input should only contain the necessary information such as the sequences definition, delay number, VW range, etc. A conceptual overview of our solution is shown in Figure 4.

Our tool, *SVAGen*, is an automatic engine that generates a set of assertions based on the templates provided as inputs. The only input parameter that the user needs to update is the “Assertion Description” part, which has a format as in Figure 5. The label name, reset signal, sequence definition, *d*, *DL*, *DH*, and assertion type is specified in the input description file.

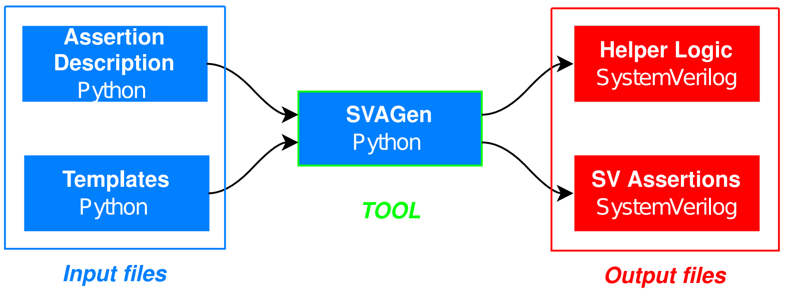
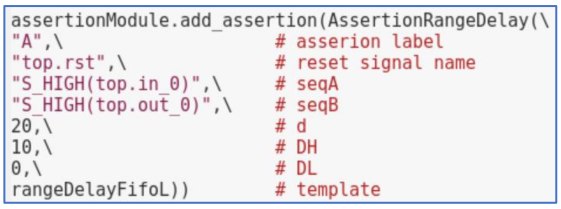


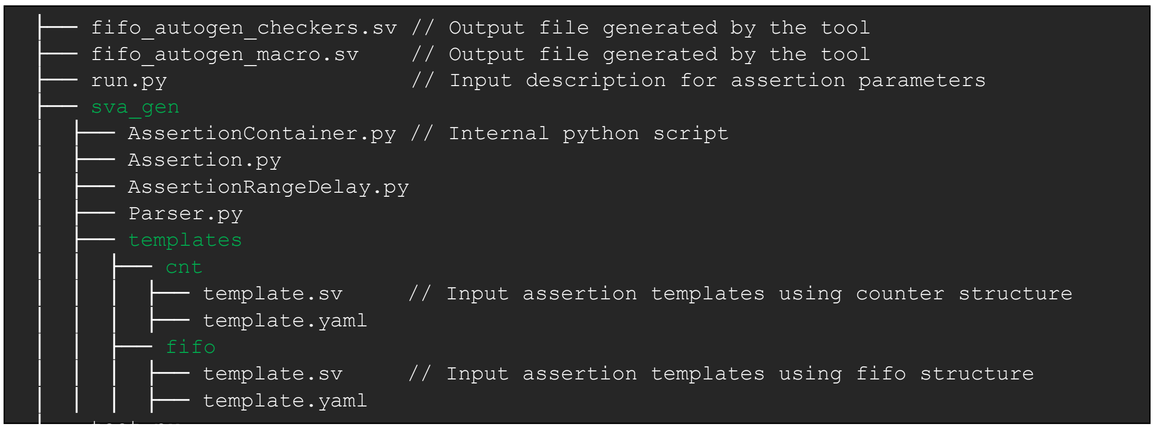
Figure 4. High level overview of *SVAGen* tool



```
assertionModule.add_assertion(AssertionRangeDelay(\
"A",\                       # asserion label
"top.rst",\                 # reset signal name
"S_HIGH(top.in_0)",\        # seqA
"S_HIGH(top.out_0)",\       # seqB
20,\                        # d
10,\                        # DH
0,\                         # DL
rangeDelayFifoL))           # template
```

Figure 5. Input assertion description format

Second part of the input is the template. It is a skeleton to generate the assertions shown in Figure 3. The automatic engine take the template and plug in the input parameters specified in the input description file, and generate the output assertion files. A directory structure of the tool and the input/output files can be found in Figure 6.



```
├── fifo_autogen_checkers.sv // Output file generated by the tool
├── fifo_autogen_macro.sv    // Output file generated by the tool
├── run.py                   // Input description for assertion parameters
├── sva_gen
│   ├── AssertionContainer.py // Internal python script
│   ├── Assertion.py
│   ├── AssertionRangeDelay.py
│   ├── Parser.py
│   ├── templates
│   │   ├── cnt
│   │   │   ├── template.sv     // Input assertion templates using counter structure
│   │   │   ├── template.yaml
│   │   ├── fifo
│   │   │   ├── template.sv     // Input assertion templates using fifo structure
│   │   │   ├── template.yaml
```

Figure 6. Directory structure of the tool with the input/output files

During the development, we experimented with templates with different implementations and size of FIFO and counters. This feature has been exploited to evaluate the best architectures and to do some design/verification space exploration of the tool. Furthermore, a Python base class for assertions is supplied as input too and it is possible to extend it to enrich each assertion with more functionalities. For example, gathering functional coverage which will be useful in simulation, and it can be easily added by extending the base class. The tool not only generates assertions, but it also allows the user to add different types of helper logic or new SVAs by changing the input template file. Both SVA and helper logic output files produced by *SVAGen* can then be passed as inputs to formal and simulation tools. In this way, engineers can save time and boost productivity by adopting *SVAGen* to automatically produce SVA and then test or assess several implementations. To make it more scalable, we also plan to add a Domain Specific Language (DSL) in the future.

IV. VALIDATION STRATEGY

In this section we focus on how we validated such an infrastructure. Firstly, we chose a DUT which is suitable for doing dynamic timing check. DUT structure and the tool environment is described in the following section. Secondly, we use a commercial formal tool to run and prove the generated assertions. Output waveform from the formal tool is checked and explained. Finally, we conducted a performance measurement on the formal converging time. The steps for this measurement and the results are discussed.

A. DUT

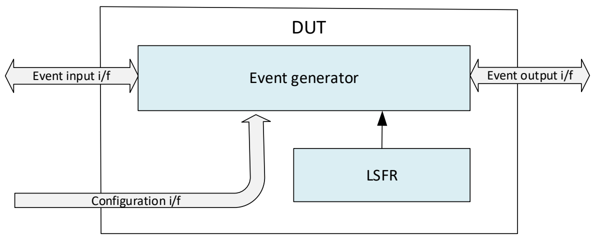


Figure 7. Overview of DUT block

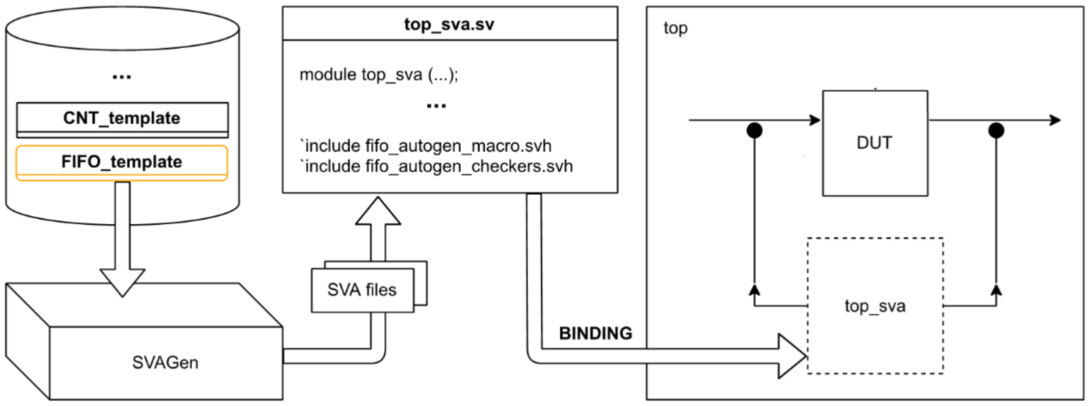


Figure 8. System overview comprehending the tool *SVAGen* and the flow at top-level.

Figure 7 shows an overview of the DUT, which is a configurable event generator. On input side, the device has an event input interface which receives multiple level triggered or edge triggered events, and a simple AMBA configuration interface. On the output side there is a similar event output interface. Based on the configuration, the device can generate a modified version of certain events. It can be delayed or logically combined with other events, etc. There is also an internal LSFR block which generates a randomized delay margin, and can add some degree of uncertainty to the output events. This highly configurable device provides us a solid and versatile baseline to challenge our tool and its autogenerated assertion.

B. *Tool flow overview*

Figure 8 represents the top-level flow of using *SVAGen*. First of all, we modify the input description file, and choose one of the templates of the assertions. Then, we run *SVAGen* to generate the SVA files where assertions, assumptions and sequences are defined. After that, the autogenerated SVA files are merged into an assertion module and then bound onto the DUT. By doing this, we ensure that assertions are running on signals from the DUT top-level. In this paper we only describe the validation process of the FIFO template. The counter template is validated too but not described here, due to the page limitation.

C. *Waveforms for a use case scenario*

To validate the assertions that are generated by our *SVAGen* tool, we used a commercial formal tool to run on the output assertions. And since we also generate the cover property from *SVAGen*, we were able to obtain some waves from the formal tool when the cover property is covered. Figure 9 depicts a small portion of one trace which is derived from the run. Dynamic delay *cfg_d* is getting re-configured at run-time, while the parameters defining the boundaries of the validity window are static, which is 3. In this case, if the input sequence (*Seq_IN*) is triggered, the output sequence (*Seq_OUT*) is expected to be triggered within a range of [*cfg_d* – *3* : *cfg_d* + *3*] clock cycles. Therefore, within this context, the VW has a width of 6 clock cycles. The portrayed part of the trace shows four different cases. Three out of four input sequences share the same *cfg_d* = 13, and the fourth has the *cfg_d* = 56. In all these cases, the *Seq_IN* and *Seq_OUT* are triggered within the correct timing frame. And the tool did not find any counter example and proved the properties without any problem.

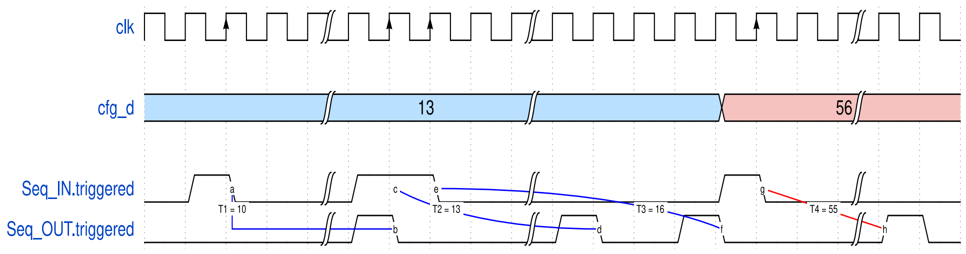


Figure 9. Trace derived from the run of a commercial formal tool.

D. Performance measurement

Performance is an important aspect of the formal based assertions. If the assertions cannot be fully proved within a reasonable amount of time, they will not be useful for our verification purpose. Our goal is to test the limit of these generated assertions and find out ways to improve the performance in terms of converging time.

We started out from an RTL implementation which supports a configurable delay up to 1k clock cycles. And we used the FIFO template to generate both assertions. The formal tool went timed-out after the predefined time-out period of 4 hours, with neither a counterexample nor a full proof.

We then tried than a few things to improve this. On the second iteration, we assumed a range of 100 clock cycles within the 1k clock cycles. This setup means the delay is still configurable, but we know approximately where the possible values are, e.g. between [801:900]. With this approach the proof was achieved in few minutes.

This gives us an idea of using a *divide-et-impera* method to prove the whole dynamic range. We split up the 1k clock cycles into ten intervals of 100 clock cycles. And we used a set of assertions to prove each interval individually.

To do this, we added an extra layer in *SVAGen* to automatically produce the assumptions which specify the interval that our configuration falls in. Depending on how many intervals, we also generate the same amount of assertion set to cover each and every interval.

Our formal tool showed good result for this. After we divide the 1k clock cycles into 10 intervals, they have all been proved within short amount of time. The interval with lower range, e.g [0:100] has a faster converging time than the higher range, e.g. [900:1000], which is also expected. We also experimented with different intervals of 10, 20, 50 and 100. The result shows that in general smaller interval range is more friendly for formal tools in converging time, but too small interval actually increased the total converging time, likely due to the overhead in each run.

To sum up the results, we show the numbers from our experiment in Table 1. The results are obtained using a FIFO template, with a maximum dynmaic timing range of 1000 clock cycles. The results show the converging time for each interval with interval size 10, 20, 50 and 100. We can see that among all the measurement, interval size of 20 has best performance in converging time. While for the other intervals, the total amount of converging time are all within reasonable range. Spliting the whole run in multiple tasks has improved the performance of *SVAGen* output assertions. It also allows the tool to parallerize the run. The only drawback of this approach is the number of assumptions and assertions are increased significantly. But since everything is autogenerated and packed into a few output files, this is not visible to the user and therefore of little importance.

Due to the time limitation, we did not try to uncover the maximum upper bound which our tool has problem to converge, even with the *divide-et-impera* approach. One reason is that DUT plays a major role in converging time. Different implementation of the RTL will result in very different upper bound limit. The formal tool also has been constantly improving in its performance. A maximum limit number is different from case to case, and will not hold as the tool itself develops. For our work, we have validated that the generated assertions can be used for checking reasonably long timing requirements. And we have shown a method which can further improve the efficiency of our generated assertion checks.

Interval			10	Interval			20	Interval			50	Interval			100
Range		Time (sec)		Range		Time (sec)		Range		Time (sec)		Range		Time (sec)	
Min	Max	A_D	A_INV	Min	Max	A_D	A_INV	Min	Max	A_D	A_INV	Min	Max	A_D	A_INV
0	10	1	0,2	0	20	1,3	1,1	0	50	1,7	2,3	0	100	4,4	3,8
11	20	1	1	21	40	1,4	1,2	51	100	2,6	4,2	101	200	12,4	13,6
21	30	1	1,6	41	60	1,4	1,5	101	150	7,6	6,8	201	300	42,6	36,2
	...				...				...				...		
971	980	27,5	12,6	941	960	13,9	24,5	851	900	101,5	203,5	701	800	334,4	230,4
981	990	28,2	11,3	961	980	12,6	15,7	901	950	56,9	47,6	801	900	136,6	283,4
991	1000	11,6	13,8	981	1000	16,3	16,7	951	1000	32,4	27,7	900	1000	126,3	48,2
Total		1352	843			390,8	505,8			675,67	973,67			981	1313,8

Table 1 Results obtained from the analysis

V. CONCLUSION

In this paper, we presented a generalized solution for common timing check requirements in block level verification. We discussed how the limitation caused by SVA syntax can be overcome by decoupling the dynamic and static components. Adding invariant and cover properties complete the check of the logical conditions. The assertion-writing process is automated into a Python based tool that generate these assertions with only the necessary input paramters. The split of the input parameter file and template file allows the user to further develop the assertions or implement other requirement. The generated assertion files are modularized which makes them easy to scale. When a project has many requirements for this kind of timing check, *SVAGen* decrease the assertion development time significantly. The outcome is well-written assertions that are uniform in format, pre-verified and ready to be deployed in the projects.

We have also tested the generated assertion in both simulation and formal tools and both showed good results. For formal tool we also looked into improving the converging of the generated assertions. *SVAGen* has been deployed in several customer projects, and the feedback from the customers is very positive. The tool is considered to be easy to use, flexible and producing robust timing checks that satisfy a wide range of verification requirements.

SVAGen will be released as an open-source project within the next months on SyoSil website at the following link [4].

VI. FUTURE DEVELOPMENT

Due to the flexibity of the *SVAGen*, it can be extended easily with more functionalities. One example could be to integrate the functional coverage model in the template and let the formal tool report coverage automatically. Another example could be the creation of more focused templates to produce targeted helper logic to reduce the COI, so that we can check timing requirements with wider range.

To ease the configuration process of *SVAGen* even further for a specific DUT then instead of instantiating Python classes directly a DSL for the configuration could be introduced. The DSL would then capture the meta data involved (label name, reset signal, *d*, *DL*, *DH*, assertion type etc.) and then *SVAGen* instantiates the correct Python class. This will also eliminate the requirements towards Python for the users.

VII. REFERENCE

[1] IEEE Standard for SystemVerilog- Unified Hardware Design,Specification, and Verification Language
Revision of IEEE Std 1800-2012

[2] SVA: The Power of Assertions in SystemVerilog ISBN 3319071394
Authors: Eduard Cerny, Surrendra Dudani, John Havlicek, Dmitry Korchemny

[3] Formal Verification: An Essential Toolkit for Modern VLSI Design ISBN 978-0-12-800727-3
Authors: Erik Seligman, Tom Schubert and M V Achutha Kiran Kumar

[4] https://www.syosil.com/resources/open-source-software