A Powerful Development Environment and its Fully Integrated Vera based System Verification Environment, for the Highly Parametric STBus Communication System

Giuseppe FALCONERI
Nizar RHOMDANE
Houssem BDIOUI
ST Microelectronics

Sal TIRALONGO
Nicolas MEUNIER
Synopsys Professional Services
Francoise CASAUBIEILH
Synopsys Verification Group

ABSTRACT

This paper presents an innovative methodology and platform to automate the architecture exploration, configuration, verification and implementation of a STBus[1] based communication backbone for SoCs. In particular the paper will focus on the STBus System Verification Environment, based on Vera, and its integration within the "STBus GenKit".

The "STBus GenKit" allows getting the maximum benefit from the STBus highly parametric features, giving a powerful instrument for a quick convergence of critical design constraints and reducing the STBus design effort. STBus GenKit allows analyzing, in an early stage of the design flow, the power, the area and the performance of any STBus topology and configuration. Thus, it helps in getting quickly to a trade-off of those constraints for any SoC application.

"STBus GenKit" is an in-house tool where the backbone design entry and configuration are accomplished through a user-friendly GUI. Still within a unique framework, all the steps from the system level (TLM and BCA) analysis to the gate level implementation are accomplished. The RTL generation and verification is automated thanks to the Synopsys coretools suite (coreBuilder, coreConsultant and coreAssembler) and Vera based verification environment STBus-SVE (System Verification Environment). The gate level synthesis follows a similar approach being based on Synopsys Design compiler and is automated thanks to coreAssember. In the platform several add-on tools are offered for each step of the

[1] STMicroelectronics proprietary on-chip bus

design flow. The final quality of the design is ensured by properties and protocol automatic check capability.

The objective of the new verification methodology is to yield a production gain. This gain is achieved utilizing pre-verified SVE based on complex protocol, higher abstraction level, ability to interact with other SVEs, modular independent components, built-in functional coverage, built-in error injection, automated generation of highly configurable RTL and SVE, built-in path to SystemC modeling (TLM, BCA). These features and much more will be described in the paper.

Table of Contents

Table of Figures

1. Introduction

The increasing complexity of SoCs these days leads to higher performance requirements. One of the key elements to insure this is related to the on chip communication system. To address performance issues, STMicroelectronics developed its own on-chip bus, “STBus”. STBus is a communication micro network that ensures high bandwidth and low latency communication.

Hence, such a high performance on-chip bus is very complex and highly configurable. Its integration requires a huge effort from the SoCs integrators in terms of implementation and verification steps. “Time to market”, as we all know, is one of the most challenging business issues.

The solution presented in this paper addresses aspects of verification, since the implementation has been widely discussed in previous papers. In particular, we address how the Vera Reference Verification Methodology (RVM) combined with a highly reusable environment provided within the STBus GenKit and coreAssembler reduces the "STBus based system" time to design. The main goal of the verification solution presented here is to provide a fully automated verification flow. It must allow setup of the verification environment in less than one day, including test cases and run scripts. It must provide state of the art verification techniques such as constrainable random stimuli, code, functional and assertion coverage, dynamic protocol check. This environment must also be easily customizable for any project specific need.

- Section 2 briefly presents the STBus protocol.
- Section 3 is about the overall STBus automated specification to implementation flow.
- Section 4 focuses on the STBus verification environment: its highly configurable architecture and main features.
- Section 5 shows how the STBus verification environment is integrated into the overall specification to implementation flow.
- Finally section 6 gives some experimental results of this fully automated process.

2. STBus Overview

2.1. STBus Protocol

The STBus [2] is a set of protocols, interfaces and components defined to implement the communication network of digital systems such as microcontrollers for different applications (set-top box, digital camera, MPEG decoder, GPS). The STBus protocol consists of three different types (namely Type I, Type II and Type III). Each one is associated with a different interface and capable of differing performance levels:

- *Type I* is a simple synchronous handshake protocol with a limited set of available command types suitable for register access and slow peripherals.
- *Type II* is more efficient than type I. Type II supports split transactions and pipelining. The transaction set includes read/write operation with different sizes (up to 64 bytes) and also specific operations like Read/Modify/Write and Swap. Transactions may also be grouped together into *chunks* to ensure allocation of the

slave, ensuring no interruption of the data stream. It is especially suited for External Memory controllers. A limitation of this protocol is that the traffic must be ordered.
- *Type III* is the most efficient. In fact, it adds support for out-of-order transactions and asymmetric communication (length of request packet different from the length of response packet) on top of what is already provided by Type II. CPUs, multi-channel DMAs and DDR controllers can therefore use it.

2.2. STBus Interconnect description

The main purpose of the STBUS interconnect IP is to address multiple-IP interconnection by providing an efficient point-to-point connection between two ends. The first end initiates a transaction to a given end that responds to the latter transaction. Both ends are implemented separately as STBUS initiators on one end, and STBUS targets on the other end. Thus, both ends work independently of each other.

The STBus architecture is modular and allows master and slaves of any protocol type and data size to communicate through the use of appropriate type/size converters (see example from Figure 2).
A wide variety of arbitration policies is also available to help system integrators meet initiators/system requirements. These include bandwidth limitation, latency arbitration, LRU, priority-based arbitration and others. A dedicated block, the node, is responsible for the arbitration and routing.

The STBUS interconnect implements arbitration and routing. The arbitration is necessary to ensure that the bandwidth associated with each initiator port is respected. It is round-robin arbitration. Routing ensures proper transfer of the STBUS transactions from an initiator to a target. The routing technique relies on address decoding; hence each STBUS target has its own memory map.

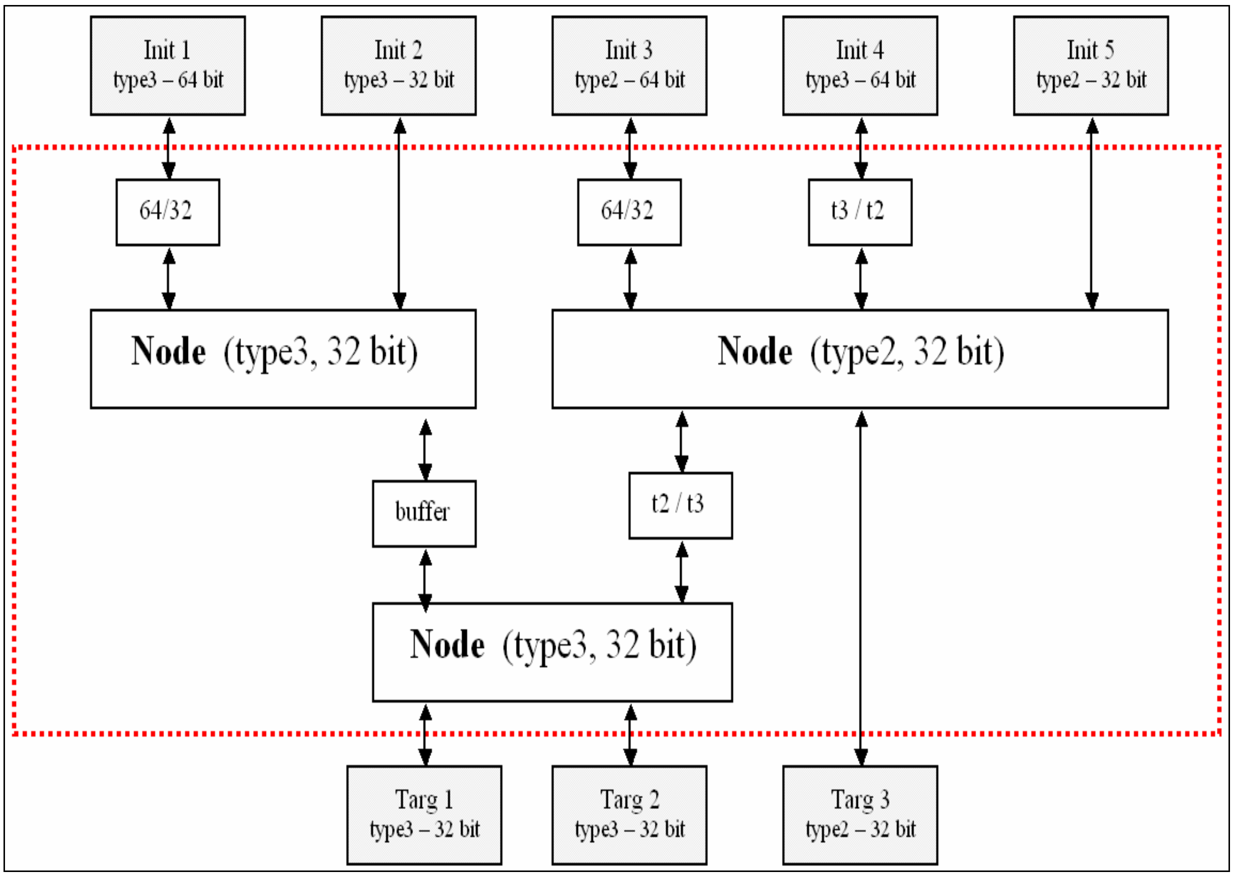


Figure 1. Example of an STBus interconnect

3. STBus GenKit Methodology

The STBus GenKit Methodology follows the top-down approach and covers all the abstraction levels from the functional specification until the gate level. All the people involved in the development of an STBus based SoC (architects, system level design engineers, designers, verification engineers and back-end engineers. etc) each contribute to step of the flow, which is shown in the following diagram:

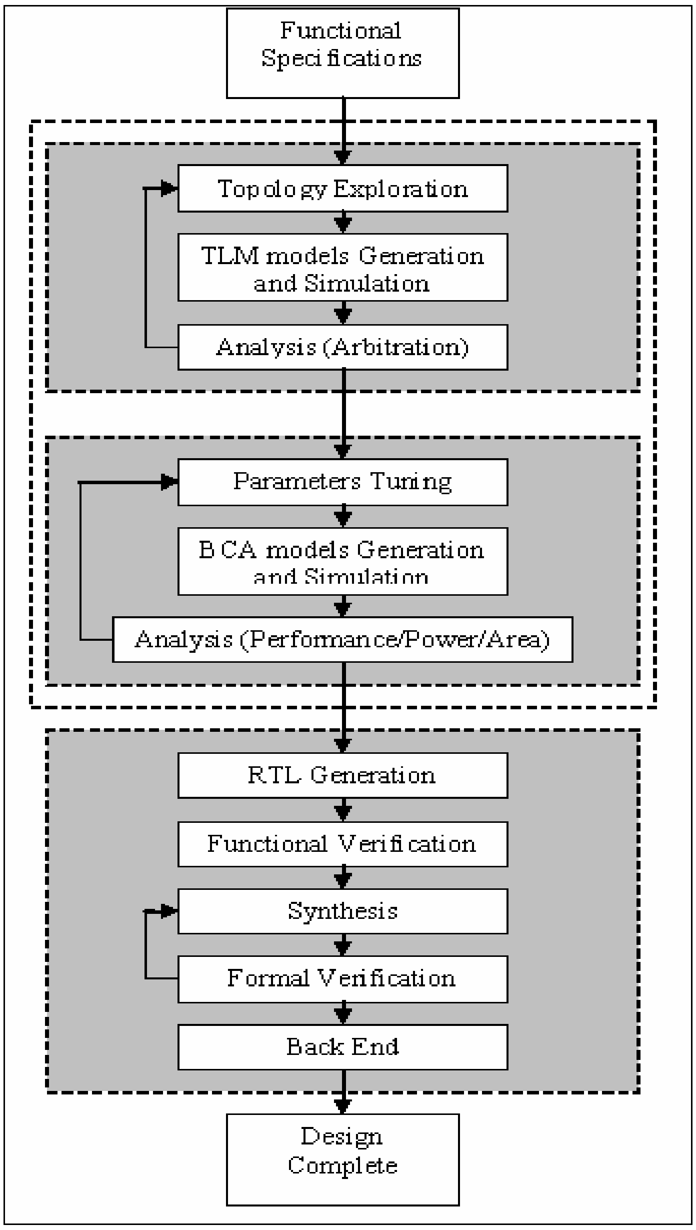


Figure 2. STBus GenKit methodology

The entry point of this flow is the SoC interconnect functional specifications where the features of the IPs to be used is defined as well as the memory map information and the requirements in term of internal communications (maximum tolerated latencies, minimum required bandwidths, etc.).

Following these requirements, the first step is to explore the eventual topologies to find a trade-off between the desired performance and the chip complexity. At this stage early software development is also started. The major constraints are the simulation speed. The SystemC TLM offers 10k times faster simulations than RTL. This allows exploration of a large number of topologies to be used, especially in terms of arbitration policies.

The interconnect architects will then proceed with the configuration and sizing of the validated topology. For this part of the process the SystemC BCA presents the most efficient view of the design. The behavior accuracy with the RTL models is around 100% while the simulation speeds are 10 to 100 times faster. The STBus interconnect blocks parameters are refined, thanks to a significant number of model generation and simulation loops. Accurate performance analysis (e.g., latency, throughput, and bandwidth) allows finding the optimum

parameters. Power consumption and area occupancy analysis offer estimation that permit to get this information at a very early stage of the design flow and not after the synthesis step that could require a long time. This allows achieving a very quick trade-off between performances, power consumption and area occupancy of the configured on chip interconnect.
Once the interconnect architecture exploration is completed, thanks to the system level part, the GUI also provides the TCL scripts to drive Synopsys' coreAssembler to generate the RTL view according to the defined topology and configuration and go ahead with the synthesis flow. The use of Synopsys' coreAssembler and the Synopsys synthesis flow has already been presented in a paper at SNUG [6].

Simultaneously with the RTL view generation, a functional verification environment is built. It checks correctness of the netlist obtained out of the system level flow. This is where the SVE plug-in enters into action to bring an unprecedented efficiency to accomplish this activity. This is achieved by utilizing a configurable and reusable verification approach. For example, the handling of multiple initiators and multiple targets, the support of multi-clock designs and the most important, the automation of the environment setup.

The integrator can then carry on with the synthesis, taking into consideration the target technology and the physical and timing constraints. Formal verification allows validating the matching of the generated netlist with the equivalent RTL. The design is now complete and can be transferred to the back-end engineers for placement, routing, chip assembly.

The interconnect development gain obtained with this flow is impressive: from scratch to gates in two months instead of one year with a classical flow with an entry point at the RTL level and no automation. A set-top-box chip was used as a test case and the results are detailed in the experimental section.

4. STBus Interconnect Verification Environment

4.1. Overall verification environment architecture

With Vera 6.2 (onward), Synopsys provides customers with Vera foundation classes that were derived from best-in-class Reference Verification Methodology (RVM). These classes are aimed at helping verification engineers to rapidly develop verification environments. The key focus is to have an environment able to be reused and to address other languages (for instance SystemC TLM).

In RVM, the ultimate advantage lies in its random tests. The tests generated by the RVM environment are designed in such a manner that all the transactions are random in nature. By constraining the test cases, one can write directed test cases as well. Thus, in order to write a directed piece of test, the only thing required is to constrain the environment such that a directed test pattern is generated.

The traditional approach is to rely on building directed environments. Hence, the DUT was driven with a known sequence of input stimulus. One could then predict the expected results and compare them against the simulated output.

This traditional methodology is no longer feasible for large designs. The traditional approach uses manual generation of the stimuli that is used to observe design behavior. Therefore, as the functionality and size of the designs increase, it becomes difficult to hit the various data and functional aspects of the design so as to ascertain the exact response of the design. In addition, directed tests are not able to catch obscure defects due to features that no one happens to think of. The randomization feature of this methodology causes random stimulus to be generated and helps in generating cases that can sometimes help in reaching cases that can never be reached by the directed approach. This methodology allows self-checking of the DUT by randomizing configurations and sequences of input stimulus and having a reference model.

The STBus SVE is designed at transaction level, up to the Bus Functional Models which convert transactions into pin level activity on the HDL interface and vice versa. With such a transaction level architecture, the STBus SVE can be reused to verify the TLM view of an STBus interconnect. In fact, the only modification required to the STBus SVE designed for the RTL, is to replace the BFMs located into the initiators and targets by a specific binding to the TLM view of the DUT. The rest of the environment is kept unchanged. The protocol checker is not used anymore since it is in charge of checking the STBus protocol at pin level.

Using RVM to implement the STBus SVE minimized effort, by using the existing foundation classes, and maximized quality by implementing state of the art methodology.

4.2. Verification Components

The STBus SVE is a set of independent verification components that can be assembled together in order to fulfill major requirements dictated by STBus-based testbenches. Main components of the STBus SVE, as shown in Figure 3 are:

- Initiator
- Target
- IP configuration
- System Configuration
- Top environment where components are stitched together
- Scheduler
- Scoreboard
- Functional coverage module
- Protocol checker

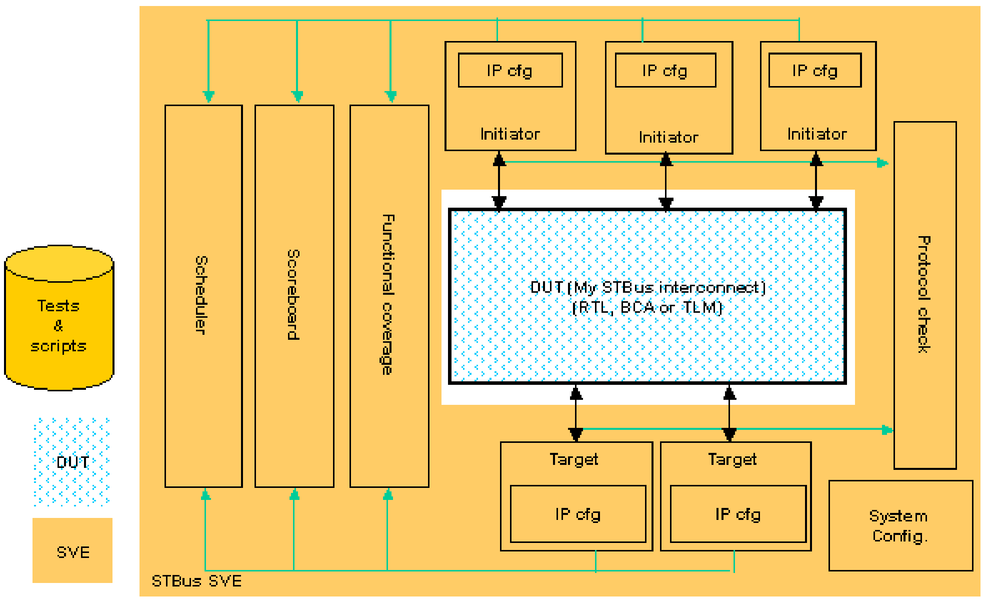


Figure 3. STBus SVE components and architecture

These components have to be highly configurable in order to support all flavors of STBus interconnects. They are all generic components except the top environment itself in which they are all instantiated. Here is a brief description for each of them:

1. **Initiator** is in charge of
   - Generating STBus transactions
   - Sending corresponding requests to the bus through its Bus Functional Model
   - Monitoring responses and
   - Passing this information to scoreboard and functional coverage modules
2. **Target** is in charge of
   - Monitoring the bus
   - Converting this activity into STBus transactions through its Bus Functional Model
   - Generating and sending responses
   - Passing this information to scoreboard and functional coverage modules
3. **IP Configuration** contains all information relative to a given IP, initiator or target:
   - Protocol type (I, II or III)
   - Data bus size
   - Pipeline depth
   - Supported pins
   - Pointer to system configuration
4. **System Configuration** contains all parameters describing the system at top level:
   - Number of initiators and targets

- Overall address map
- Interconnect internal register file mirror
- Simulation controls such as timeouts or maximum number of transactions

5. **Top environment** where all bricks are assembled together in order to build the exact verification environment required by a given STBus interconnect. It's major functions are:
   - Build the environment by instantiating and connecting the right number of initiators, targets with the scoreboard and functional coverage modules
   - Start all transactors, i.e. generators and monitors
   - Waits for end of test conditions
   - Stops all transactors in a orderly fashion
   - Drains DUT of any buffered data
   - Reports success or failure of the simulation run
6. **Scheduler** is in charge of controlling all initiators and targets from a system point of view. It can activate or deactivate, individually or globally, a set of transactors. This action can occur on an event or after a given time.

As shown in this section, there are three extra components, coverage module, scoreboard and protocol checker which are addressed in next sections.

4.3. Functional coverage (transactions and protocol)

The STBus SVE contains a **functional coverage** module in charge of ensuring that all coverage points defined into the verification plan have been covered. It focuses on two main areas: routing and latency.

- Routing coverage: checks that all initiators sent all types of transactions (measures on opcodes, transaction ids, store mechanisms) to all their possible targets. Measures are also made to ensure that all initiators sent invalid requests (to unmapped addresses) without breaking the system. This coverage group is DUT dependant since the interconnect may not support all possible connections from any initiator to all targets.
- Latency: checks that all possible types of stress to the bus have been applied by each initiator (at request) and each target (at response). This stress corresponds to the latency inserted by the initiators or targets before sending their packets. This latency is measured in number of clock cycles.

The functional coverage module is also used as a global monitor, which tracks all activity through the interconnect during a simulation. This is why it can provide bandwidth information (average number of bytes sent per second, measured over the entire simulation run) for each transactor and each test case.

4.4. Self-check

Since the STBus SVE is a constrainable random environment, we are not in a traditional context with directed test where results are predictable. The environment must be able to check on the fly that the DUT is behaving properly. This self-check is performed through three axes:

- From inside the initiator or the target which reconstruct transactions from the activity observed on the bus: this reconstruction mechanism will raise errors if some unexpected behavior is observed. An example for an initiator may be the reception of a response when no request was sent.
- Through the usage of assertions used as **protocol checkers:** the STBus protocol is described in a set of formal rules gathered in the so-called **protocol checker**. These rules are bound to each port of the interconnect and raise errors as soon as rules are violated. Note that coverage is measured on these assertions so that we ensure that all protocol scenarios have been tested. The implementation of this protocol checker by a set of formal assertions allows reuse of this rule set by a formal tool to perform static verification.
- With the **scoreboard:** this module is designed at the transaction level. It checks to see that a request sent by an initiator reaches the target it is addressing (**routing check**). It also checks that data flowing over the interconnect is not corrupted (**data integrity check**). In other words it checks that all bytes of a data word sent by an initiator for a Store operation reaches its target. The same check is performed on the response data sent by a target on a Load operation.

4.5. Testing flavors

In the previous sections, we have seen the different bricks that build the constrainable random STBus SVE. Test cases are a set of constraints applied over this environment in order to force this environment to perform interesting scenarios. The STBus SVE contains a set of generic test cases that can be run on all possible STBus interconnects:

- Concurrent tests: where all initiators and targets generate traffic in parallel
- Sequential tests: where all initiators except one are stopped, this initiator accesses sequentially each of its possible targets. When it has generated enough traffic, it is stopped and next initiator is started
- Register tests: which access internal registers of the interconnect in a write then read mode
- Unitary tests in between 1 single initiator towards 1 single target
- Error injection tests: which generate transactions toward unmapped addresses
- Protocol specific test: which force specific mechanisms such as ‘write posting’ or ‘store and forward’
- Saturation tests: an example is a test where requests are sent at high rate while responses are very slow in order to fill-in the STBus interconnect internal pipeline

All these tests have been written with a generic approach in mind in order to be reusable over all environments without any modification. They can be disabled from non-regression databases when the specific features of the STBus protocol they verify are not used in the Design Under Test.

5. Reuse

5.1. Overall coreTools approach

The main principles in creating and delivering a reusable verification environment can be summarized as follows:

- Easily configurable to fit different applications
- Designed for use in multiple technologies
- Thorough commenting
- Well designed verification environments and suites
- Robust scripts
- Easy to use and friendly implementation interface
- Good documentation

The methodology described here addresses most of the above.

Tools and processes are needed that capture the design information in a consistent, easy to communicate form and make it easy to integrate modules into a design when the original designer is not available. The integration flow has to ensure quality of results, ease of use and tool support over multiple versions or licenses. The package has to provide all the necessary views and necessary tests across all possible parameter values.

The solution can be found in the new methodology offered by Synopsys' suite of reuse tools. The innovation of coreTools is to address all the previous needs: it has to provide industry leading tools that enable users to create, package, deploy, integrate and assemble configurable soft IP.

CoreTools support flow customization. Steps (activities) that already exist within default CoreTools activity flow can be modified to perform extra work. Additionally, you may add new steps to the default flow (link to other tools). These customizations are described in plug-ins, which are tcl files. Plug-ins manipulate specifically instrumented files, that are customized by use of dedicated coreTools functions. Once the files are annotated and the plug-in is ready there are all loaded together in a single .kb file. Through the development and usage of plug-ins a developer can provide interfaces to any tools tailored to a specific design flow. The integration of the STBusSVE is made using plug-ins.

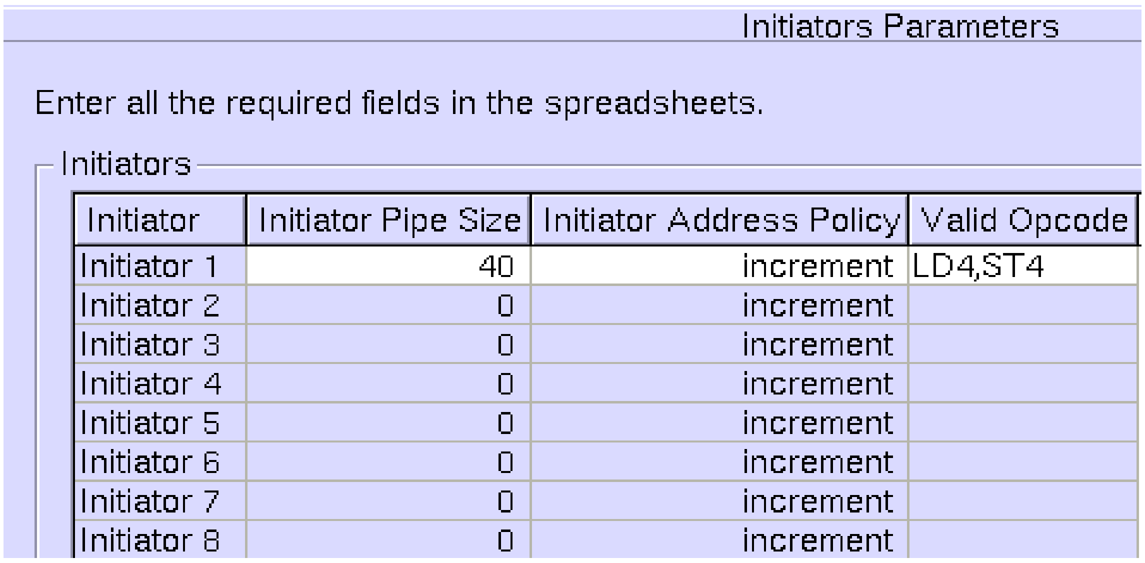
5.2. STBus System Verification Environment Plug-in

As explained above STBusSVE provides a set of Vera components. They can be utilized to develop and simulate Testbenches that would check the functionality of any STBus Interconnects. The traditional approach to design a Testbench is to select components among this STBusSVE "library" to develop it from scratch. This does not guarantee that STBusSVE features will be used properly. This approach is extremely time consuming.

It is much more efficient to use automation to assemble these Testbench bricks together, and to take care of parameterizable tests simulations. Consequently, two plug-ins are built. One creates the STBusSVE. The other one manages simulation environment and launches tests.

The first plug-in is responsible for hiding the Testbench implementation details from users and taking them smoothly from STBusGenKit to a "ready to simulate" environment.

STBus Interconnect designers who want to build a STBusSVE for their DUT have only to deal with the following GUI:


Initiators Parameters

Enter all the required fields in the spreadsheets.

Initiators

Initiator	Initiator Pipe Size	Initiator Address Policy	Valid Opcode
Initiator 1	40	increment	LD4,ST4
Initiator 2	0	increment	
Initiator 3	0	increment	
Initiator 4	0	increment	
Initiator 5	0	increment	
Initiator 6	0	increment	
Initiator 7	0	increment	
Initiator 8	0	increment	

Figure 4. Snapshot of "Create Vera testbench" plug-in in coreAssembler

Little information has to be entered manually via the GUI, (i.e., mainly the configuration parameters come from STBusGenKit). As STBusGenKit generates the RTL code of the node, it also generates a set of files that contains design information in a coreTools readable format. Only parameters that cannot be extracted out of the scripts have to be passed via the STBusSVE plug-in coreAssembler GUI. Parameters generated by the STBusGenKit are, for instance, either for an initiator its type or for a clock its period and phase.

From a user's perspective, no details are visible, and developing a STBusSVE is very fast. To obtain such an ease of use, two issues had to be resolved.

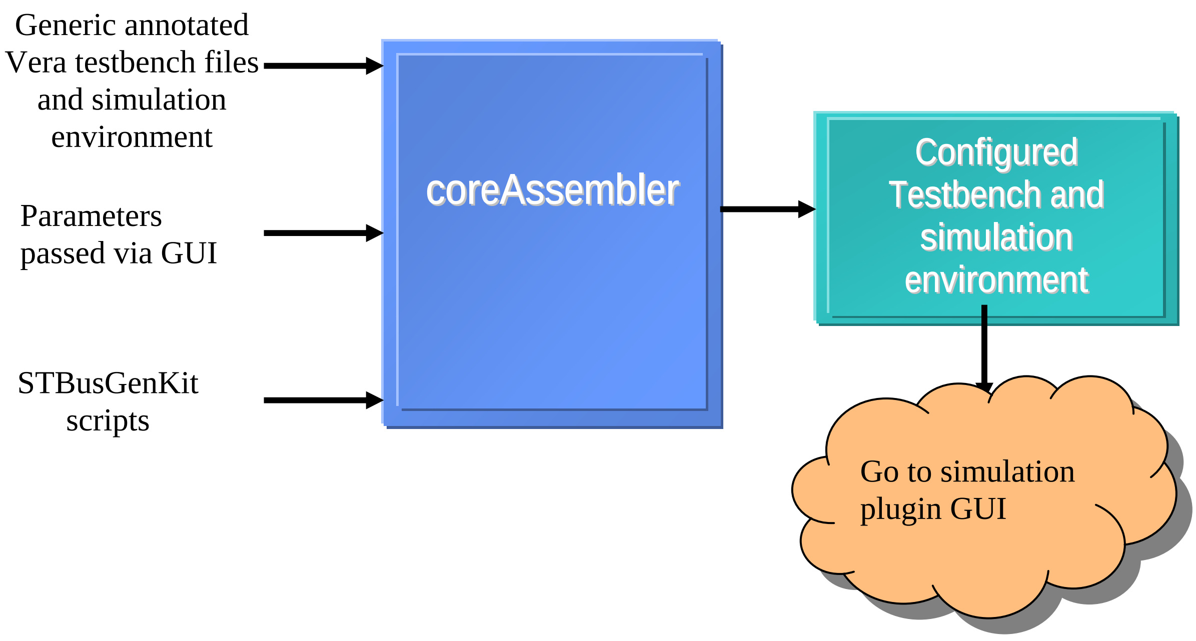
1. The integration of coreAssembler into the STBusGenKit flow.
2. It was necessary to create a plug-in that manages transparently all STBusSVE files.

The plug-in is actually responsible for configuring the testbench at different levels:

- Vera-HDL interface files
- Vera top level files
- Functional coverage files
- SystemVerilog Assertions binding files
- Tests files

The specific structure of the STBusSVE brings here a significant benefit in terms of plug-in development. Many STBusSVE components are generic and can be reused as is. So, most of the parameterizable features are contained within the environment declaration. Then

information is propagated to the rest of STBusSVE components. And as a result, very few files need to be annotated.
A key point of the plug-in is multiple clocks DUT support. The plug-in is in charge of generating adequate HDL top level code to issue the clocks, whose parameters are read directly from the STBusGenKit automatically generated scripts. There is only one flow for single clock and multiple clock designs.


Generic annotated Vera testbench files and simulation environment
Parameters passed via GUI
STBusGenKit scripts
coreAssembler
Configured Testbench and simulation environment
Go to simulation plugin GUI

Once the design information is collected, the STBusSVE files have to be customized properly to match the DUT specifications. The first step is to modify those files in a highly parametric way, so that all the different configurations of the blocks can be chosen by simply setting some parameters. This is the annotation phase. This consists in adding some lines to the source Vera files. They are interpreted as comments by simulators and compilers, but are commands for the Core Tools. Then with respect to parameters value, these lines can be:

- Inserted or removed from final testbench files

```
…………
cfgi[1-INIT_1ST_IDX] = new(
   // reuse-pragma startSub [IncludeIf [get_activity_parameter STBusVIP endianess]==1 %subText]
     LITTLE_ENDIAN,
     // reuse-pragma startSub [IncludeIf [get_activity_parameter STBusVIP endianess]==0 %subText]
     BIG_ENDIAN,
………
```

- Or alternatively it can be customized directly into text

```
 ………
// reuse-pragma startSub [format " GNT_ON_REQ_%d," [get_activity_parameter STBusVIP init_1_rgnt_prtcl]]
………
```

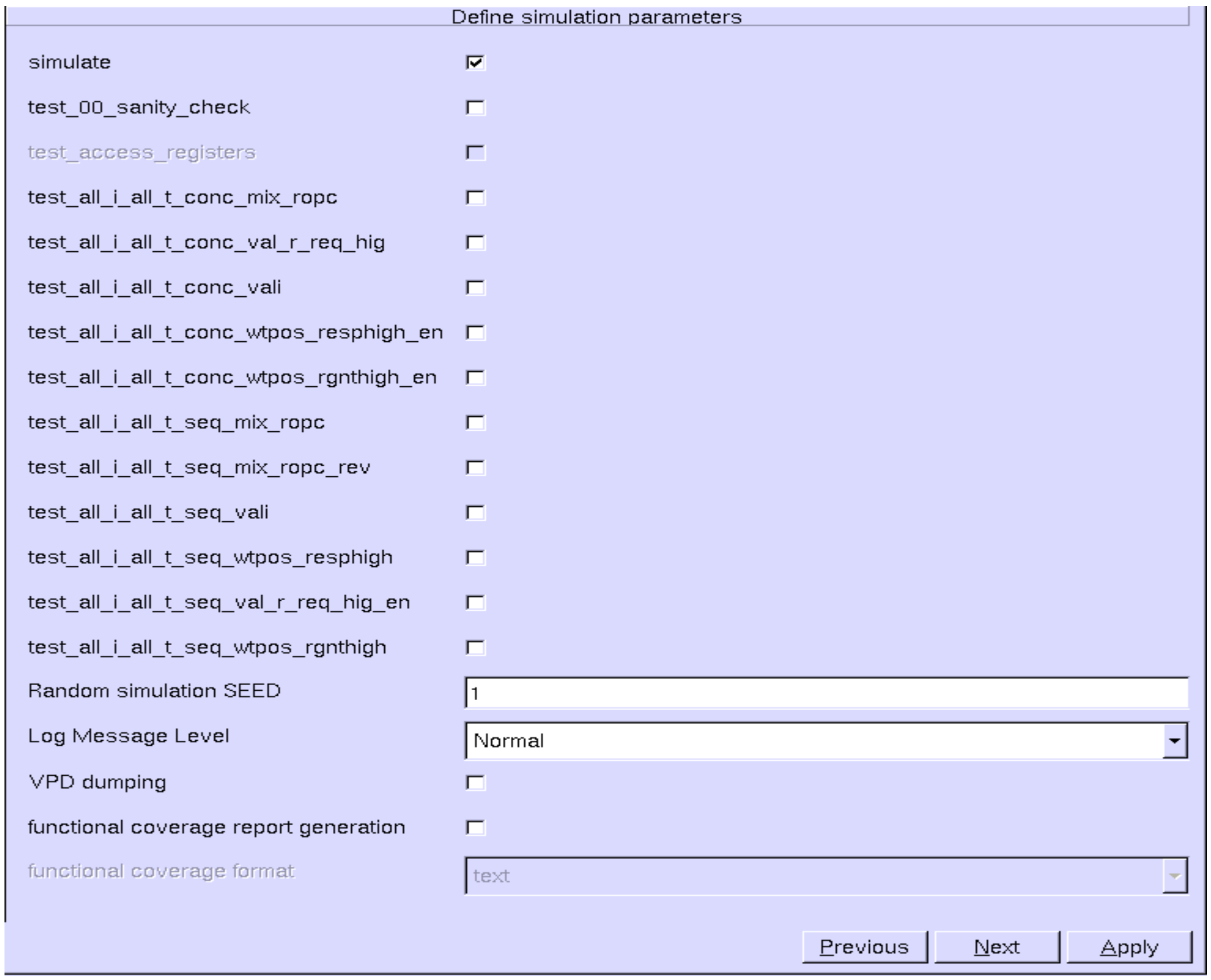
When the plug-in is launched, it replaces all those annotated lines by the correctly configured ones, automatically and transparently.
One feature that takes great advantage of this flow is functional coverage. Functional coverage is one of the biggest burdens of the plug-in development. This powerful feature does not make any sense if the coverage goals are not set appropriately. In turn coverage goals are highly dependent on the DUT parameters. Consequently the plug-in is in charge of collecting design information by use of dedicated coreTools tcl functions. That information might not be directly issued by the STBusGenKit, but may rather be computed out of the information provided by those scripts.

```
proc STBusVIPProc {} {
  …
  set nb_init_loop [get_activity_parameter STBusVIP nb_init]
  …
```

For instance the number of accessible targets for each initiator and the number of memory arrays for each target are used to calculate the number of accessible memory arrays for each initiator. In turn this is used to set Functional Coverage goals.

Automation brings here a few advantages for users. Designers may not be aware of the functional coverage feature’s details, they might miss the necessity of properly setting the coverage goals. Also they could have trouble figuring out what section of the code has to be modified and where to get the data to modify it appropriately. This short example shows the impact, in terms of time savings, of the coreTools flow usage associated with the STBusSVE.

Users build the testbench only once. Then they can use the second plug-in to run as many simulations as needed without having to regenerate the testbench. This second plug-in is in charge of creating and populating the simulation directory. Like the first plug-in it gathers information from the STBusGenKit and a specific GUI. It then customizes a bunch of setup files, Makefiles, tests files and results files.



simulate	☑
test_00_sanity_check	☐
test_access_registers	☐
test_all_i_all_t_conc_mix_ropc	☐
test_all_i_all_t_conc_val_r_req_hig	☐
test_all_i_all_t_conc_vali	☐
test_all_i_all_t_conc_wtpos_resphigh_en	☐
test_all_i_all_t_conc_wtpos_rgnthigh_en	☐
test_all_i_all_t_seq_mix_ropc	☐
test_all_i_all_t_seq_mix_ropc_rev	☐
test_all_i_all_t_seq_vali	☐
test_all_i_all_t_seq_wtpos_resphigh	☐
test_all_i_all_t_seq_val_r_req_hig_en	☐
test_all_i_all_t_seq_wtpos_rgnthigh	☐
Random simulation SEED	1
Log Message Level	Normal
VPD dumping	☐
functional coverage report generation	☐
functional coverage format	text

Figure 5. Snapshot of "Run Vera" plug-in in coreAssembler

The plug-in actually splits the process into several steps. It takes advantage of the STBusSVE simulation Makefiles to allow the user to go through the useful steps smoothly and independently.
For instance, users can:

- Perform cleans (Vera, HDL, results files)
- Specify execution options (LSF queues)
- Customize tests to be run
- Compile Vera
- Compile RTL
- Visualize Waveforms
- Generate functional coverage reports (txt, html)

Obviously, all these features are available by use of the STBusSVE; the plug-in make them more users friendly.
In addition to this, a big advantage of using the plug-in here is that it allows users to access only features that are relevant. As a matter of fact, The STBusSVE permits the checking of all STBus features. Nevertheless users may want to generate designs that do not support all of them. Consequently, when a DUT, for instance, does not support features checked by specific tests, users are not allowed to select these tests. Actually the tests could either fail or hang, and this is not what users want. All the available tests are then listed for the user. They just

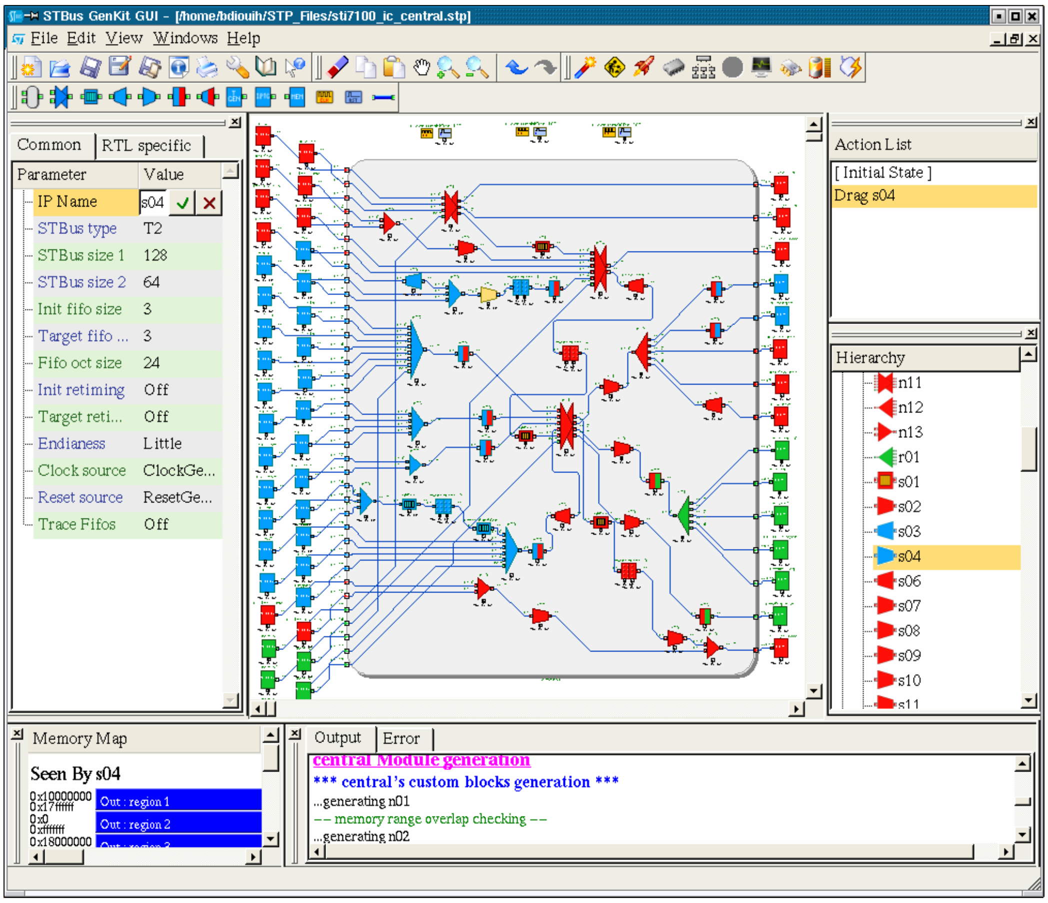
have to pick those they want from the GUI.. Users then do not need to become familiar with any simulation scripts or have to bother remembering command line and tool specific switches.

Moreover, the plug-in adds an additional level of Makefile features. It keeps track of tests that are compiled and also for parameterizable tests. It keeps track of the configuration that was compiled. So, with respect to all user input the plug-in determines an optimized list of files to recompile. This adds another level of incremental compile. This is all transparent to users and associated with the basic Makefiles, this is a huge gain in terms of compilation time.

Eventually, all simulation results' files are also accessible through coreAssembler GUI. Tests log files and functional coverage reports are all presented in an easy to read way. Users do not have to know where the files are stored and how to open them. All relevant information is put together within the same place once the simulation is done.

6. Experimental results

The STBus GenKit is currently in use in about fifteen projects within STMicroelectronics Consumer and Microcontrollers division. Among those an example is a set-top-box chip for terrestrial, satellite, cable or DSL STBs conforming to all the major operator requirements worldwide. It is able to support two simple definition TVs, audio/video decoding, de-scrambling, all in one low cost chip. It embeds two CPUs and a DSP. The major implementation constraints are to have a significant cost reduction and performance upgrade in dual TV applications.
The SystemC TLM phases allowed defining the STBus interconnect topology in two weeks' time. The sizing and parameters tuning was obtained thanks to the SystemC BCA phase in one-month time with all the performance metrics, power and area constraints respected. The RTL was generated using the implementation step and a validated gate level netlist was obtained in less than one month. The total development time was about two and a half man months. During the placement and routing few issues were reported which allowed a fast first mask generation. Based on previous available data it would have taken approximately twelve man months to complete the development of the same interconnect utilizing the old manual flow.



Parameter	Value
IP Name	s04
STBus type	T2
STBus size 1	128
STBus size 2	64
Init fifo size	3
Target fifo ...	3
Fifo oct size	24
Init retiming	Off
Target reti...	Off
Endianess	Little
Clock source	ClockGe...
Reset source	ResetGe...
Trace Fifos	Off

[Initial State]
Drag s04

Seen By s04

0x10000000 0x17ffffff	Out : region 1
0x0 0xffffffff	Out : region 2
0x18000000	Out : region 3

```
central Module generation
*** central's custom blocks generation ***
...generating n01
-- memory range overlap checking --
...generating n02
```

Figure 6. Set-top-box STBus interconnect developed using STBus GenKit GUI

7. Conclusion

In this paper, we have described an innovative methodology and platform to automate the architecture exploration, configuration, verification and implementation of a STBus[2] based communication system for SoCs. In particular, the paper has focused on the STBus System Verification Environment, based on Vera and coreAssembler, and its integration within the "STBus GenKit", pointing out its advantages compared to the traditional design flow methodology.

It is evident how the new flow makes the verification and implementation of a complex system, such as the STBus interconnect, quite simple, thanks to the user friendly graphical interface, and safe from errors, due to the automation of a variety of tasks previously performed by hands. Moreover, the generation of all the different work environments, such as functional verification, synthesis and timing analysis environments, is automatic and configuration dependent, so that all the environments are generated taking into account the selected architecture of the STBus.

Another advantage of this flow is the possibility of directly interfacing with the System Level Design flow, through which it's possible to perform architectural exploration and tune the system configuration to get the best performance, so as to start the silicon implementation directly basing on the optimum architecture.

[2] STMicroelectronics proprietary on-chip bus

It is worth pointing out that this methodology, developed around the STBus interconnect, can be used for any IPs and sub-systems, so it shouldn't be seen as something specific for a well defined application, but rather as a generic philosophy to be applied to IPs and sub-systems' design.

8. Acknowledgements

Last but not least, we would like to point out that the achievement of the results described in this paper has been possible thanks to the fruitful cooperation between the STMicroelectronics OCCS (On Chip Communication System) team and Synopsys SPS (Synopsys Professional Services) and VG (Verification Group) divisions, whose contribution to the development of the new flow has been fundamental to shorten the time and to increase the quality of the work.

9. References

[1] T. Grotker, S. Liao, G. Martin and S. Swan, "System design with *SystemC*", Kluwer Academic Publishers, 2002.
[2] *"STBus Functional Specs"*, STMicroelectronics, public web support site, http://www.stmcu.com/inchtml-pages-STBus_intro.html, STMicroelectronics, April 2003.
[3] Trolltech website : http://www.trolltech.com
[4] Open SystemC Initiative : http://www.systemc.org
[5] R. Zafalon, V. Zaccaria, A. Bona, "System Level Power Modeling and Simulation of High-End Industrial Network-on-Chip", IEEE DATE 04, February 2004, Paris, France.
[6] Pistritto C., Falconeri G., Tiralongo S., STMicroelectronics puts SoC integration into fast forward. In Proceedings of SNUG2003.
[7] Reference Verification Methodology User Guide Version 6.2
[8] Reference Verification Methodology Training Material
[9] Vera^R User Guide Version 6.3
[10] Verifying the ST BusSwitch using Vera and RVM. SNUG Boston 2004.

10. Glossary of acronyms

AOP	Aspect Oriented Programming
BFM	Bus Functional Model
CRT	Constraint Random Testing
DUT	Design Under Test
DW	Synopsys Design Ware
OOP	Object Oriented Programming
SOC	System On a Chip
RVM	Reference Verification Methodology
VIP	Verification Intellectual Property
SVE	System Verification Environment