A scalable verification methodology using VERA

Venkat Rajaraman, Narayanan Vydianathan, Khosrow Hajikhani

Networking and Security Group, Sun Microsystems

venkataraman.rajaraman@sun.com
narayanan.vydianathan@sun.com
khosrow.hajikhani@sun.com

ABSTRACT

SUN's networking products group is working on a new high performance networking ASIC to meet the growing demand of bandwidth and packet processing requirements. For such complex SoC design, there is a definite need for doing verification at multiple levels of hierarchy at module, chip and system level. Thus far, there has been too much of overhead in building and maintaining various environments at these multiple levels. The hierarchical verification is built using reusable verification components. The verification environment at various levels is consistent and built dynamically. This is achieved through modularizing the verification components and reusing it at different levels. There is a Response checker for each module which computes the expected response. Every packet in the environment is uniquely identified by a token. Mailboxes are used to pass the tokens from one response checker to another. More details about the environment reuse and dynamic creation using VERA is presented in this paper.

1.0 Introduction

Most companies now realize that functional verification of complex ASICs has become an inefficient, unwieldy and incomplete process. So there is a tremendous need for verification reuse - from module level to chip level to system level. A strategy for verification reuse is modularization. Typically verification environment is viewed as large, unstructured mass of code that surround the design being verified (the “DUV”). Part of this reason is that the verification environment is generally built up incrementally, without a focus on the overall needs of the verification environment. In this paper we discuss the various aspects of verification methodology that enables reuse.

2.0 Hierarchical Verification Methodology

The verification methodology is based on unified Vera based environment consisting of drivers and checkers. For module level verification, the environment use light-weight vera behavioral models to generate (or respond to) transactions (called drivers) to the module under test. These transactions are monitored by vera checkers. There will be one checker per module under test. Verification is done in a hierarchical fashion i.e. verify extensively at module-level and reuse the checkers developed for module-level verification at chip-level as well. The following sections give an overview on the verification mechanism.

2.1 Tokens and Mailboxes

The verification methodology is adapted for verifying one of the SUN's networking ASICs where the predominant data path is the packet flow. The verification environment maintains a separate packet database containing the packet header and data, checker state variables and the flow state variables. At any point of time the environment will have enough data stored in the database so that the expected packets can be reconstructed. This allows the checkers to dynamically reconstruct expected data and check them against the data received on the interface.

Every packet in the verification environment is identified by a unique token. Tokens are assigned to every packet by the Packet Generator. The data structure for the token is predefined. Tokens are passed from one Checker to the other using Mailboxes. The token flow is predefined based on the packet data path within ASIC i.e. token flow is identical to the data path flow inside the RTL.

For a module-level test environment the token flow originates from one of the drivers and it is passed to the module response checker. For the chip-level verification environment, the token flow originates at the packet generator and it gets retired at the last packet checker.

In order to simplify the token passing between drivers and checkers, the flow is controlled by a set of global classes and tasks. By using these methods the module verification engineer is relieved from burden of knowing where to get the tokens from and where to send the tokens to.

A global task which looks at the current configuration of the environment is called by the drivers/checkers to determine which mailbox the data is retrieved from. In case, if a particular module has multiple input data path, then the input driver should call the global task multiple times to get the mailbox handles. Also if a checker/driver wants to put a token into a mailbox (mailbox_put), it allocates the mailbox prior to writing to it. All the mailbox handles are defined in a global class and initialized to a negative number. This allows the drivers/checkers to determine whether the mailbox is active prior to use.

2.2 Response checkers and drivers

The functions of a checker block are summarized below:-

1. Module Response Checker connect to all the active interface of the module ("DUV") and verifies if the interface signals adhere to the predefined interface protocol. Any discrepancy is flagged as an error. (Protocol Check)

2. Checkers also verify the packet data on the interface against the expected data. (Data Integrity Check).

Operations performed by a Checker:

• Wait for a token from the preceding Checker or driver.

• Access the Shadow-class registers and CAM-RAM entries to predict the behavior of the RTL. A "shadow copy" of the programmable registers is maintained separately. Whenever a register or memory in this programmable address space is updated, the "shadow copy" also gets automatically updated. This copy is extremely useful in retrieving the register contents on the fly by drivers and/or checkers, without actually causing a bus cycle. There are global methods available to read and write into the shadow space.

• Rebuild the expected packet based on the received token.

• Perform the Protocol Check and Data Integrity check on the module interfaces.

• Update the packet database, if needed.

• Pass the token to the down-stream checker.

• Checkers may optionally enable VCA (Value Change Alert) on the interface control signals.

Checkers may also perform read/write operations to a file to keep track of internal checker events (Checker Debug). The checkers communicate with one-another using tokens.

Drivers are behavioral vera models used for module-level verification. A driver mimics a given interface for the module under test. The functions performed by a driver is summarized in the following statements:-

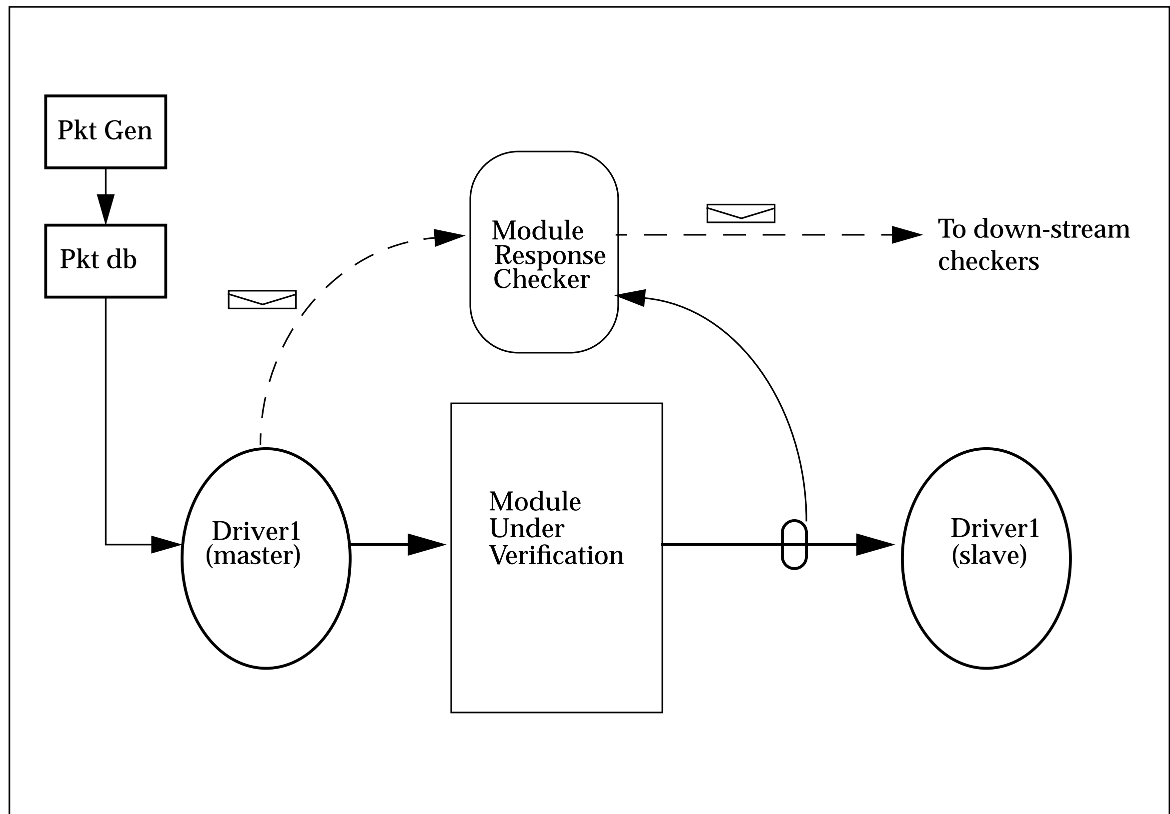
1. Transfer the headers/packets generated by the packet generator to the module under test as defined by the interface protocol.

2. Passes the token to the downstream Checker.

2.3 Module verification environment

The purpose of doing module level verification is multi fold. Firstly, when a given module is being tested in all likelihood the neighboring blocks to which this module needs to interface may not be ready to be tested. Secondly, generating all corner case test conditions at module level is much more simpler if the test controls interfaces of the module-level rather than the chip's interfaces at top level. And lastly, targeting tests at module level cuts down on simulation time. In this mode, "drivers" provides stimulus for the module under test and "checkers" look for expected result from the module under test as shown in Figure 2-1.

Figure 2-1 Module Level Verification



The Master driver, drives the Module Under Verification (DUV) to run specific cycles. The Module Response Checker computes the expected activity and compares it against which is being driven by DUV.

1. Module checker and driver are two independent entities.

2. In the full-chip RTL simulation environment, only the checkers will be instantiated.

3. There should be no direct link between a module checker and its corresponding module driver except through mailboxes. Otherwise, it would be very difficult to de-link them when the checkers have to be instantiated at chip level.

4. Module checkers get tokens from upstream checkers or upstream drivers. A checker receives tokens from upstream driver or checker seamlessly.

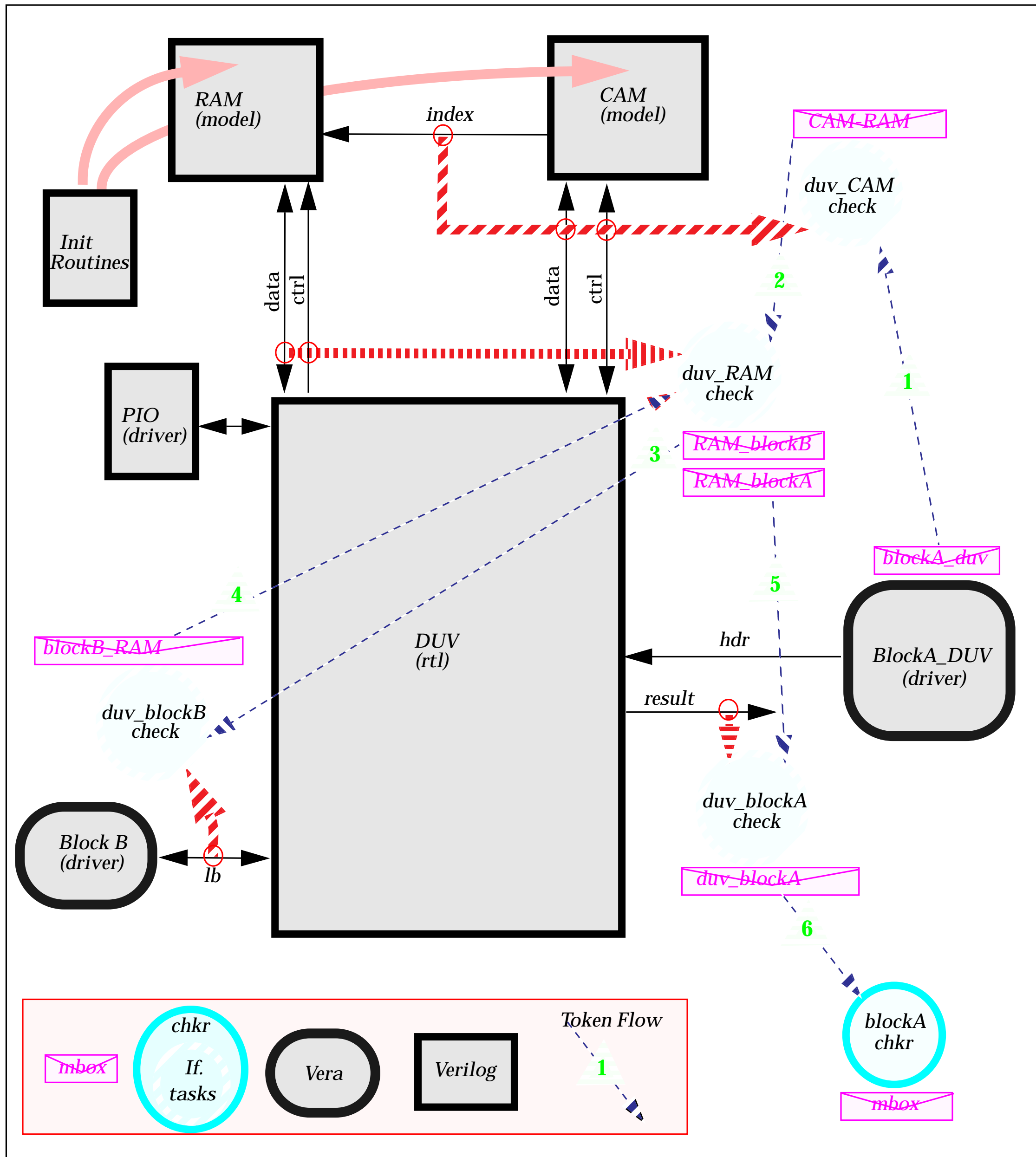
5. Module checkers may receive tokens from more than one upstream driver or checker.

6. Module checkers pass tokens to downstream checkers and may pass multiple tokens to different downstream checkers, based on data path.

7. Module tests, instantiate the various drivers and checkers required for module verification and also controls the drivers and checkers for finer control of the module interface.

A module level token flow mechanism is illustrated in Figure 2-2. The DUV is interfacing with BlockA and BlockB in addition to memories. The BlockA driver is a master driver which initiates a transaction to the DUV. A token is passed to the DUV checker, which corresponds to that transaction. This checker class consists of multiple interface check tasks. This verifies if the DUVs expected behavior at the CAM-RAM interface and also finally checks if the result which is returned back to BlockA is correct.

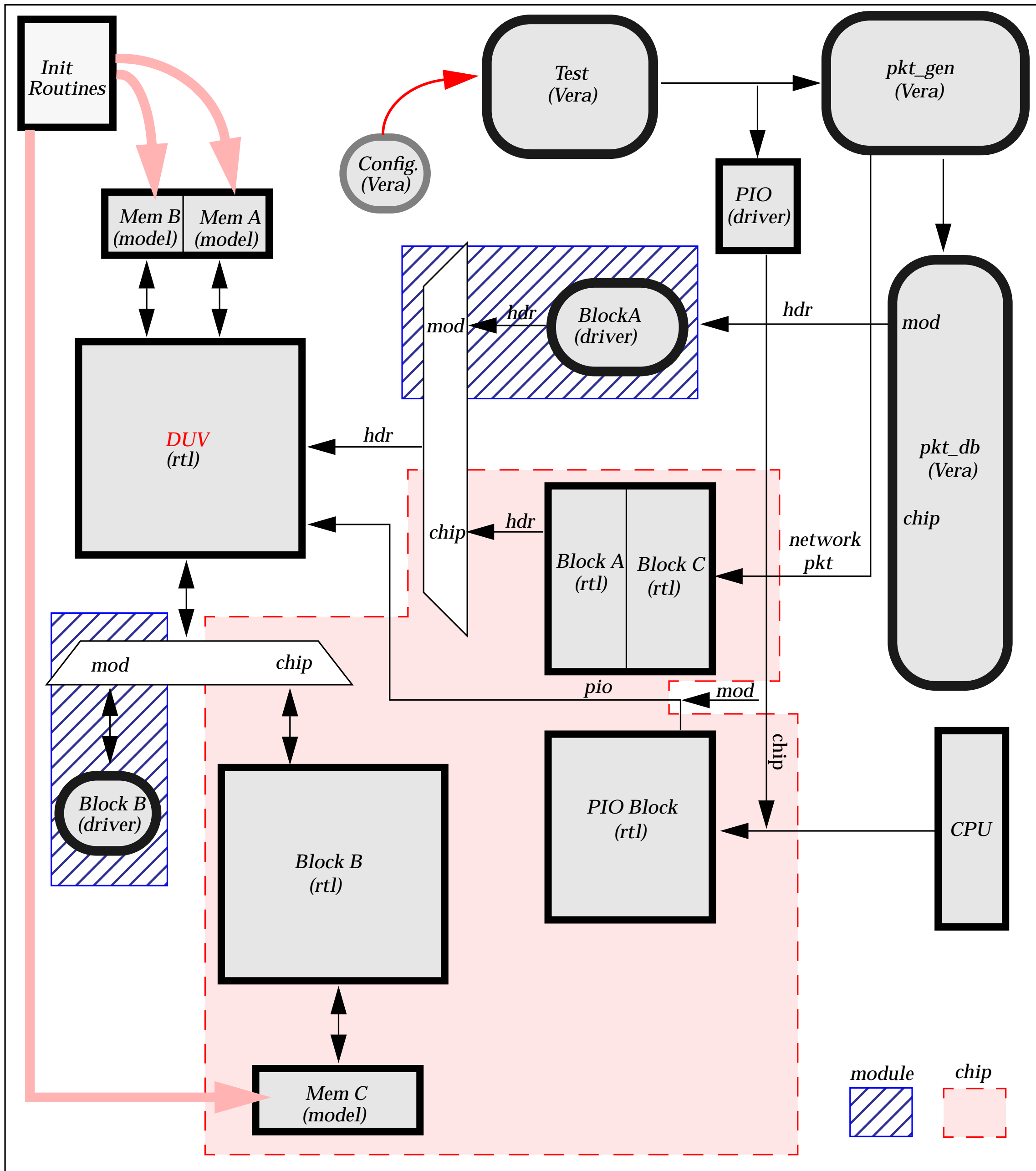
Figure 2-2 Module Level Token Flow



2.4 Dynamic Configuration

There is a common verification environment for module, chip and system level verification. The “blocks” required for the creation of the module-level or chip-level or system level verification will be dynamically decided based on a user defined configuration file. The configuration file decides which of the blocks are RTL and which are drivers. In a typical module-level verification scenario, the module under test is defined as RTL and rest of the modules interfacing to it are defined as drivers. In some module level test environment it is quite likely that the module under test does not need to interface to a given block, in such cases that block is replaced with a dummy place holder. Dummy blocks are Verilog modules with the output control signals “pulled” to an inactive state, inputs and data paths are left floating. Using a configuration file has yet another advantage, the checkers can be dynamically activated based on the configuration. This dynamic configuration effectively controls the instantiation of objects in the top vera program.For example, in the chip level environment the user is expected to configure all the blocks to be RTL. Hence no “drivers” will be instantiated but all the “checkers” will be activated automatically. Figure 2-3 depicts the dynamic configuration scheme. At the module level, the drivers for Block A and Block B are active. Also, the PIO driver which is used for accessing the internal registers, directly interfaces with the DUV instead of going through the PIO RTL block. At chip level, the Block A and Block B drivers are replaced by the actual RTL. The checkers are re-used at chip level.

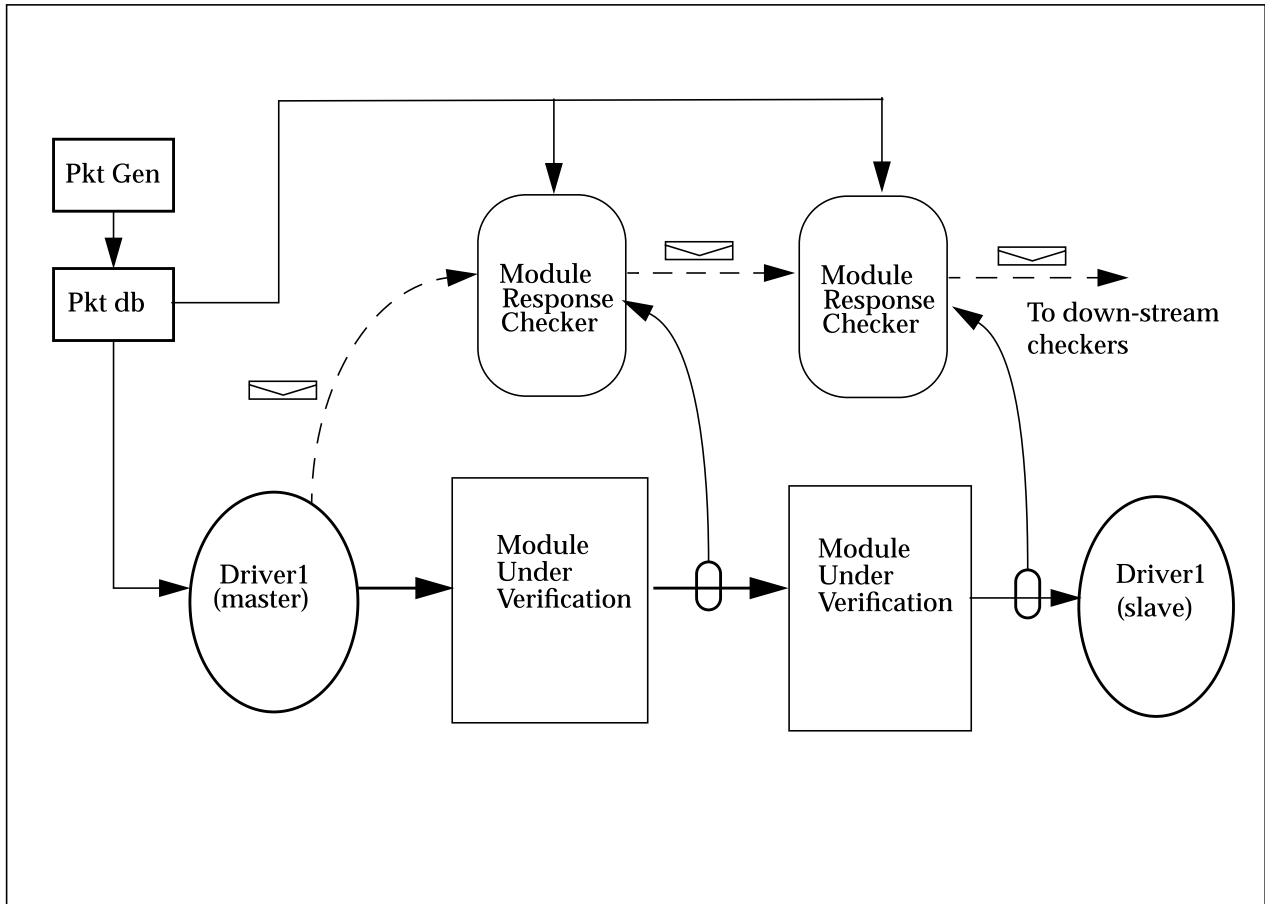
Figure 2-3 Chip Level vs. Module Level auto-configuration



2.5 Chip verification environment

Verification environment for multi-module configuration is shown in Figure 2-4, which is easily extended to chip level and system level environment. The module response checkers is re-used. This methodology is especially powerful during debugging, as the problems as it happens in a module is immediately caught by the module response checker instead of waiting for it be propagated through all the modules before showing up at the chip level interface.

Figure 2-4 Multi-module / chip level Verification



3.0 Garbage Collection

Since the environment heavily relies on multiple packet databases, it is desirable to have a mechanism to purge the used database effectively. If we are sending thousands of packets through the chip, the database of the first packet need not be kept till the last packet is sent out of the chip. To accomplish effective purging of the database the following method is suggested.

Maintain a status_db class per token with following members.

```
class status_db {
    bit[m:o] signature_reg;
    bit[m:0] error_reg;
}
```

Here m is no of checkers in the chip. Every checker sets a bit corresponding to its position when the packet is successfully checked at that checker.At the final checker at the chip interface if status_db[token].signature_reg[m:0] == m'b1;. we can remove the database corresponding to the particular token. This is accomplished by explicitly setting the object handle to null. Also the error register can be initialized to a start value and the bits could be updated when status register updates the status bit. The error_reg helps in telling in which checker a particular "token" failed. Garbage collection in Vera is automatic. In addition, elements of associate arrays which are no longer needed are explicitly de-allocated by deleting the element.

4.0 Conclusion

Comprehensive verification strategy is the key to scalable verification and verification re-use. This paper presented a verification methodology which separates the environment into two different types of reusable components, drivers and checkers. In fact, the checker functionality could be divided into protocol monitors and data integrity checkers. With this, even the protocol monitors could be re-used at various levels even if data integrity checking functionality is not needed. Also, this methodology reduces the simulation time required to find bugs, when compared to a conventional methodology. This is because the error detection on any interface is immediate. The checker keeps track of all the activity during simulation (at the transaction accuracy level) and checks the response right away. This results in more accurate error reporting with less simulation time. Table 4-1 gives a brief description of the various components of the verification methodology along with a brief description.

Table 4-1 Various hierarchical verification components.

	Language	**Re-usable at chip level?**	**Brief Description**
Driver	Vera Class	No	Used at module and/or multi-module level to initiate or respond to the transactions to/from the DUV.
Checker	Vera Class	Yes	Checks the response of the DUV. Re-used at chip level for faster error detection.
Token Manager	Vera Class	Yes	Handles the token flow for the entire ASIC. Also represents the data path of the device.
Dummy	Verilog	No	Dummy verilog blocks used for dynamic configuration.

5.0 References

1. The art of verification with Vera - Faisal I. Haque et al

2. What is Garbage Collection in Vera and How Does It work? Synopsys SolvNet article Misc-213.html dated 3/15/2000.

3. Sun Vera Style and Coding guidelines - N. Korpusik et al. Sun Proprietary internal document.

4. Synopsys Vera Home. http://www.synopsys.com/products/vera/vera.html

5. Synopsys VERA User Guide. Version 5.0. October 2001.