Automating the IEEE std. 1500 compliance verification for embedded cores

Automating the IEEE std. 1500 Compliance Verification for Embedded Cores

A. Benso, S. Di Carlo, P. Prinetto
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy.
Email: {alfredo.benso, stefano.dicarlo, paolo.prinetto}@polito.it

A. Bosio
Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier
Universit de Montpellier II / CNRS
Montpellier Cedex 5, France.
E-mail: alberto.bosio@lirmm.fr

***Abstract*—The IEEE 1500 Standard for Embedded Core Testing proposes a very effective solution for testing modern System-On-Chip (SoC). It proposes a flexible hardware test wrapper architecture, together with a Core Test Language (CTL) used to describe the implemented wrapper functionalities. Already several IP providers have announced compliance in both existing and future design blocks. In this paper we address the challenge of guaranteeing the compliance of a wrapper architecture and its CTL description to the IEEE std. 1500. This is a mandatory step to fully trust the wrapper functionalities in applying the test sequences to the core. The proposed solution aims at implementing a verification framework allowing core providers and/or integrators to automatically verify the compliancy of their products (sold or purchased) to the standard.**

I. Introduction

The race to market high-volume quality products demands a shorter design-to-manufacturing cycle, forcing System-on-Chip (SoC) designers to strongly rely on Intellectual Property (IP) cores from multiple sources [1]. The shorter time-to-volume requires faster silicon bring-up with a high degree of diagnosability [2]. This means being able to isolate each embedded core during test and debug activities. The adoption of adequate test and diagnosis strategies is therefore a major challenge in modern SoCs production.

The IEEE Standard Testability Method for Embedded Core-Based Integrated Circuits (IEEE std. 1500 [3]) addresses the specific challenges that come with testing deeply embedded reusable cores supplied by different providers, who often use different hardware description levels and mixed technologies [2] [4] [5]. It defines a comprehensive set of guidelines for building the core test infrastructure. It includes:

- A *Core Test Wrapper*: a wrapper placed around the boundaries of the core that allows accessing its testing functionalities using a standard interface and protocol (Figure 1). The wrapper is completely transparent when the core is not in test mode;
- An *Information Model*: a formal description of the IEEE std. 1500 functionalities implemented by the Core Test Wrapper. The standard supports many functionalities, some mandatory and some optional. The Information Model is the bridge between core providers and core users and facilitates the automation of test data transfer and reuse between these two entities. The Information Model is described using the IEEE std. 1450.6 Core Test Language (CTL) [6] and includes:
  - The set of wrapper's signals;
  - The wrapper communication protocol;
  - Information about test patterns.

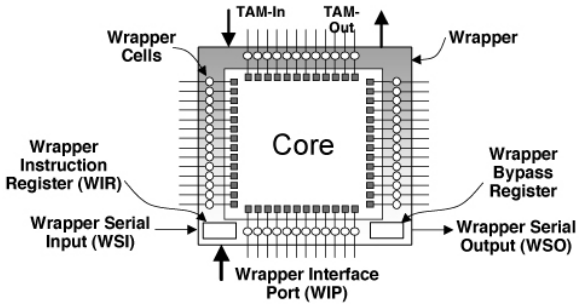

Fig. 1. IEEE std. 1500 Core Test Wrapper Architecture

Some literature presents solutions to build SoCs with IEEE std. 1500 testability features [7] [8]; nevertheless, by analyzing the standard, it is clear that implementing a fully compliant core is not a trivial task. The need to support as wide a range of embedded core test applications as possible has led to a very flexible solution. As described in [4], although a mandatory minimal set of hardware support is defined, a designer can extend the test infrastructure by creating virtually unlimited sets of registers and instruction extensions [9]. An IEEE std. 1500 compliant design is therefore exposed to a range of possible design errors that require to be early identified and fixed. A comprehensive approach to thoroughly verify the functionality of IEEE std. 1500 wrappers and wrapped cores in a SoC environment is therefore mandatory.

The problem of verifying the compliance of an IP core to the IEEE std. 1500 has been poorly addressed in literature. In [10] the authors presented an approach based on a dynamic constrained-random coverage driven verification methodology to verify the functionalities of the complete test infrastructure within a given SoC. The authors present a strategy based on a single verification module performing the full verification.

1-4244-1480-6/07/$25.00 ©2007 IEEE

The main drawbacks of the proposed solution are that, for each core, the verification module must be configured by hand and that the verification strategy, i.e. the order used to verify each aspect (rule) of the standard, is fixed. Moreover, the authors verify the SoC and wrapper functionalities without systematically addressing every single aspect (rule) of the standard.

To overcome these problems, in [11], we present a verification framework based on the use of the UML [12] language, designed to systematically address the verification of the standard. Besides providing the actual implementation of the framework, the paper focuses on the definition of an abstract model of the standard enabling core providers and/or integrators to build their custom verification environments.

This paper aims at showing how, starting from the abstract model proposed in [11], it is possible to build a verification environment for the IEEE std. 1500. In particular we will show how the functionalities provided by Specman Elite^{TM}(Cadence) [13], a commercial functional verification EDA can be used to implement such a verification environment.

We suppose the reader familiar with the basic aspects of the IEEE std. 1500. If needed, a complete description of the standard can be found in the IEEE std. 1500 official document [3].

The paper is organized as follow: Section II introduces the proposed verification environment and its architecture. Section III presents the functional verification issues related to the IEEE std. 1500, while Section IV details the adopeted solution related to the functional verification. Finally, Section V provides some experimental results and Section VI concludes the paper.

II. Architecture overview

Figure 2 shows the overall architecture of our IEEE std. 1500 Verification Environment. The verification process consists of two distinct phases named *static* and *dynamic* check, respectively.

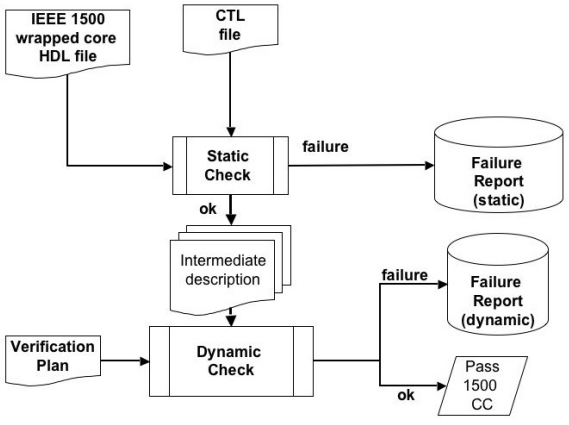

Fig. 2. IEEE std. 1500 Verification Environment Architecture Overview

The static check phase aims at verifying the correctness of the Information Model provided together with an IEEE std. 1500 compliant core (see Section I). This model is expressed using a standard language (CTL [6]) and, for this reason, verifying its correctness consists in verifying the correctness of the syntax of the CTL description. This is a well known problem in the field of programming languages and compilers. It is usually solved using special programs called lexers and parsers [14].

Several technologies allow the automatic implementation of parsers starting from the description of a formal grammar. We successfully implemented a CTL syntax analyzer by using two open-source tools: JFLEX [15] by Vern Paxson together with the parser generator CUP [16] by Scott Hudson. The resulting analyzer is able to automatically perform a syntax analysis on any type of Information Model provided together with an IEEE std. 1500 compliant core.

A failure in the static check phase means that either the CTL or HDL model of the wrapped core are affected by syntax errors. The *Failure Report* (Figure 2) is the list of these errors. In case of a positive check, the output of this phase is a collection of information extracted from the CTL and HDL code of the Core Test Wrapper (i.e., information about signals naming conventions, etc...) represented using a so called *Intermediate Description* (Figure 2), and used as input for the next verification phases. Since the static check phase is not complex, we will not provide additional details on its implementation while in the remaining of the paper we will focus on the more interesting dynamic aspects of the verification.

In order to understand the tasks to perform during the dynamic check phase we have first to recall the structure of the IEEE std. 1500 itself. The IEEE std. 1500 is composed of a set of different rules that define how an IEEE std. 1500 compliant Core Test Wrapper has to be designed [3]. The rules identify two different classes of aspects to be verified:

- *Semantic aspects*: they concern the Information Model (CTL) and can be verified without any interaction with the actual core/wrapper implementation (i.e. without the need of core/wrapper simulations);
- *Behavioral aspects*: they target the communication protocols and the behavior of the Core Test Wrapper. In general, the verification of these aspects, requires a functional simulation of the wrapper.

Semantic aspects mainly aim at identify the correct definition of the following structures in the Information Model:

- Scan structures;
- Macro definitions;
- Environments.

They can be verified by analyzing the content of the Information Model provided with the Core Test Wrapper. This is a fast and powerful way to verify at least part of the IEEE std. 1500 compliancy since it does not require any simulation of the wrapper/core itself. Although simple and fast, this analysis is not enough to guarantee the compliance w.r.t. the standard. First of all, semantic aspects are only a relatively small subset of the whole set of rules to verify. Moreover, the semantic

analysis is performed on data contained in an information model supplied by the core provider; there is no guarantee that the CTL description perfectly matches the actual hardware implementation.

Semantic aspects can be verified by resorting to the information stored in the Intermediate Description obtained as the output of the static check phase (see Figure 2). An efficient way to implement the semantic aspects verification is to store the Intermediate Description into a relational database. In this situation the verification tasks can be easily translated into a set of queries performed on the database.

Behavioral aspects are the most complex part of the standard and are the most difficult to verify. Their verification requires the simulation of the core/wrapper functionalities. Being the most important part of the verification flow, a detailed description of how behavioral aspects can be verified will be provided in the next paragraphs.

III. Dynamic functional verification to verify IEEE std. 1500 behavioral aspects

The verification of behavioral aspects is based on the functional simulation of the Core Test Wrapper and of the core itself. Simulation is the only effective approach to verify the compliancy of time-related rules, protocols, signal connections, and correct instructions implementation.

A well-known approach to perform this type of verification is the so-called *"dynamic coverage-driven constrained-random simulation functional verification"*. The term *"dynamic"* refers to the fact that the verification patterns/stimulus are generated and applied to the design by simulating/executing the design model, and the corresponding results are collected and compared against a reference/golden model. An EDA simulator is used both to compute the values of the signals during the simulation, and to compare the expected values with the reference ones.

Simple dynamic verification has a main drawback: only a subset of the possible behaviors can be verified in a time-bound simulation run. Testing all possible behaviors under every possible combination of input stimuli is, in most of the cases, an unfeasible task since the test space is too large to be fully covered in a reasonable amount of time.

To overcome this problem, the number of verification patterns applied to the wrapper has to be statistically significant but not complete. To do this, verification input patterns are generated randomly under a set of constraints, expressed as mathematical expressions limiting the set of legal values on the input signals that drive the design. In this way the simulator generates random values and constraints ensure that the generated scenarios are valid and plausible.

To further optimize this constrained-random generation, coverage-driven verification is used. Functional coverage metrics are automatically and in real-time stored in order to ascertain whether (and how effectively) a particular test verifies a given feature. This information can then be fed back into the generation process in order to drive additional verification effort towards the required goal. The coverage metrics are evaluated on coverage monitoring points defined by the user and specified in the verification plan.

The market offers a number of tools able to support this dynamic (or functional) verification methodology. The most used ones are Specman Elite™(Cadence) [13] and Vera™(Synopsys) [17]. Besides the different verification and pattern generation engines, all of them apply the verification patterns to the target design using a verification component placed around the core under analysis. The verification component, a behavioral-level module described using a proprietary verification language (*e* for Specman Elite™, *OpenVera* for Vera™, and *SystemVerilog* for SystemVerilog™), performs the constrained-random generation of the verification patterns, applies the patterns, and is directly controlled by the verification engine monitoring the current coverage reached in the verification process.

In the following paragraph we will show how to verify IEEE std. 1500 behavioral aspects using Specman Elite™and and its verification language *e*.

IV. IEEE std. 1500 Behavioral Aspects Verification using Specman Elite™

Specman Elite™ [13] is a comprehensive environment able to manage the different aspects of the verification flow of an integrated circuit: automatic generation of functional tests, data and temporal checking, functional coverage analysis, and HDL simulation control. Figure 3 sketches the overall architecture of Specman Elite™.

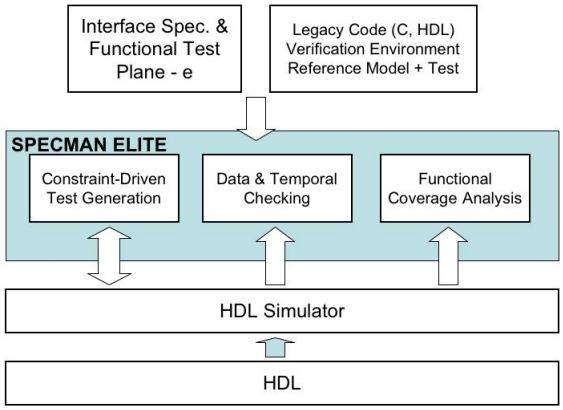

Fig. 3. Specman Overall Architecture

Specman Elite™uses the IEEE Standard *e* Functional Verification Language [18] (*e* for short) to capture behaviors defined into the specifications as well as to automatically generate tests. Powerful temporal constructs enable to capture the aspects of complex protocols for checking. The *e* language allows writing *"e Verification Cores"* (eVCs). eVCs are software modules modeling the functional behavior of the environment surrounding the target system. Specman Elite™is able to interface with HDL simulators in order to have full controllability and observability of internal signals of the device under verification. Moreover, by means of an executable

functional test plan, it automatically measures the progress of the verification identifying holes in the test coverage. In this way, the verification schedule becomes more predictable since the functional coverage is a meaningful and direct measure of the completeness of the verification.

The use of eVCs and the full internal signals controllability/observability are key elements to efficiently verify IEEE std. 1500 behavioral aspects. To efficiently apply the Specman Elite^{TM}verification approach to the dynamic verification of IEEE std. 1500 it is necessary to:

- Create a rule verification component for each IEEE std. 1500 behavioral rule. The component is in charge of generating the verification patterns applied during the simulation and checking that all the architectural and behavioral aspects of the rule are correctly implemented in the design;
- Identify the rule coverage points for the given rule. A coverage point is a Core Test Wrapper signal/register to be monitored in order to evaluate the coverage reached during the verification process.

The concept of rule coverage is very important. As stated in Section III, to reduce the complexity of the verification, the verification engine resorts to the constrained-random pattern generation. This leads to the application of a subset of the possible patterns to the system under verification and therefore to a rule coverage or compliancy level possibly lower than 100%. The challenge in verifying behavioral aspects of the IEEE std. 1500 is to write rule verification components and to identify rule coverage points that are independent from the specific core or wrapper under analysis.

Another very important issue to be considered at this point is the level of controllability and observability on the core/wrapper design. Mainly, we can distinguish between two situations: black-box design and white-box design. The difference is in the amount of available information on the core/wrapper internal structure. In a black-box approach the only available information is the I/O interface of the wrapper/core. For Intellectual Property (IP) protection the internal structure of a black-box core is unknown. A core integrator, who buys cores from different vendors, usually deals with black-box designs. On the other hand, a core designer has complete access to the core/wrapper design, and therefore deals with white-box designs.

From the IEEE std.1500 compliancy verification point of view, the difference between black-box and white-box design directly impacts the level of verification compliancy that can be achieved. In a white-box design, the internal signals of the core/wrapper can be fully controlled and/or observed and therefore all IEEE std. 1500 rules can be fully verified. On the other hand, in a black-box design, only rules (or the portions of them) that do not require direct controllability or observability of core/wrapper internal signals can be fully verified. Full IEEE std.1500 verification compliancy can only be achieved when dealing with white-box cores or with black-box cores implementing only the basic requirements of the standard.

The implemented verification environment is structured as a single eVC (e Verification Component) architecture under the recommendations of eRM^{TM}(e Reuse Methodology) [13]. The verification component is able to deal with a generic wrapped core without any additional modification. Core depended information are directly provided to the verification component by a set of *e* configuration files automatically generated starting from the information stored in the Intermidiate Descritption (see Section II). The following subsections will detail the implementation of the proposed verification environment using the *e* language.

A. IEEE std. 1500 Verification Environmente eVC Structure

This section overviews the eVC structure required to verify behavioral aspects of the IEEE std. 1500. Each aspect will be identified using the corresponding rule number contained in the standard specification document [3]. Figure 4 shows the main elements composing this verification component.

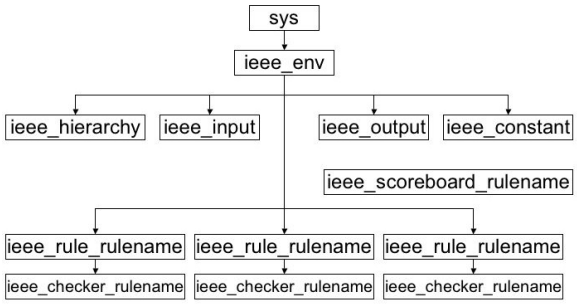

Fig. 4. IEEE std. 1500 eVC structure

Each box in Figure 4 corresponds to a Specman Elite^{TM}structure [13] implemented as an *e* file. The remaining of this section will detail the meaning of each block.

The `ieee_env` block is the highest level in the hierarchy; it extends the predefined Specman Elite^{TM}structure `sys`, in order to instantiate the verification framework, and it is in charge of defining the events required to perform the actual verification. In particular, for each rule to verify, it defines a *start* event to begin the verification process and an *end* event required to control the results of the verification process. An example of `ieee_env.e` file is reported in Figure 5. The example contains a flag checkpoint, used to stop the verification process if one of the rules under analysis is not respected.

The `ieee_input.e`, `ieee_ouput.e` and `ieee_constant.e` files contain information related to the specific Core Test Wrapper under verification. They are automatically generated during the Static Check phase (see Figure 2) starting from information contained in the Information Model and stored in the Intermediate description. They provide information used to apply the stimuli at the input signals of the wrapped core, to monitor the output signals of the wrapped core, and to store wrapped core specific information as wrapper signals names, registers size etc.

```
event start_13_1_1_a;
event end_13_1_1_a;

event start_7_2_1_c;
event end_7_2_1_c;

event start_7_2_1_b;
event end_7_2_1_b;

...
...

checkpoint: bit;
keep soft checkpoint == 0;
```

Fig. 5. IEEE std. 1500 eVC environment example

The file `ieee_hierarchy.e` contains the information about the verification plan. It defines the order in which the different IEEE std. 1500 rules has to be verified and the dependencies between the different rules. The verification plan is really important to highlight dependencies between the results of the different verification steps. Moreover, being the verification process one of the main cost factors of a modern SoC, we have to define optimal verification plans able to reduce the overall verification time.

An example of a small portion of this file is reported in Figure 6.

```
on end_10_3_1_a3 {
 emit start_10_3_1_j;
};
on end_10_3_1_j {
 if checkpoint == 1 {
 stop_run;
 } else {
 emit start_7_4_1_a;
 };
};
```

Fig. 6. IEEE std. 1500 Verification Plan Example

Each time the verification of a given rule ends i.e., the `end_rulenumber` event specified in the environment occurs (see Figure 5), the verification plan identifies the next rule verification to schedule. In case of rules dependencies, the result of a specific rule verification may modify the verification plan. As an example let us consider rule `7_4_1_a`. According to the verification plan of Figure 6 its verification starts only if rule `10_3_1_j` is correctly verified. In a different way, according to the same verification plan, rule `10_3_1_j` is verified after rule `10_3_1_a3` without checking if rule `10_3_1_a3` has been violated or not.

To conclude, each IEEE std. 1500 rule has been mapped to an *e* module named (`ieee_<rulenumber>.e`) where `rulenumber` is the number of the rule as defined in the standard. The module always includes two *e* files:

- `ieee_scoreboard_<rulenumber>.e`;
- `ieee_Checker_<rulenumber>.e`.

The scoreboard aims at checking the correctness of the rule by comparing the values observed at the output of the wrapped core with the expected one. The checker aims at checking the correctness of the timing protocol of the wrapped core.

B. IEEE std. 1500 Rules Verification Strategy

As stated in Section IV-A the most critical element of the proposed verification environment is the instantiation of the verification modules in charge of verifying the different IEEE std. 1500 rules. This task is performed by a so called *agent* implementing the verification plan defined by the `ieee_hierarchy.e` file (Figure 7). For each rule the agent instantiates two different modules:

- *Bus Functional Model (BFM)*: it generates the input stimuli and drives the signals of the wrapped core required to verify the corresponding rule;
- *Monitor*: this module comprises the Scoreboard and the Checker already defined in Section IV-A.

Finally, for each rule a set of *coverage items* are defined. Specman Elite™allows the definition of three types of coverage items to cover all the possible coverage events:

- *basic*: how many times an event occurs;
- *transition*: how many times an item changes state from a range of values to another;
- *cross*: how many times an event occurs relative to how many times other events occur.

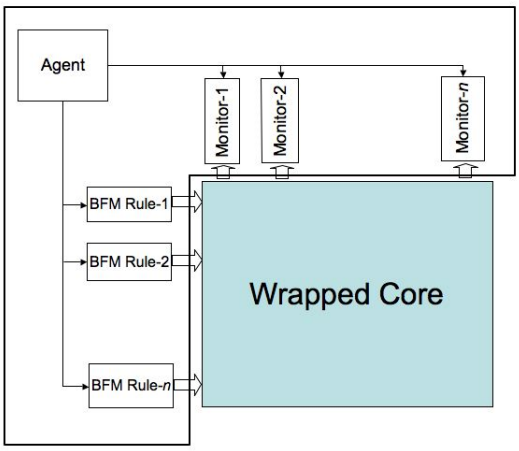

Fig. 7. Rule Verifier Architecture

To better understand the proposed verification strategy let us consider the following IEEE std. 1500 rule:

While the WS_BYPASS instruction is selected, all wrapper boundary cells that can operate in either system or test modes shall perform their system function.

To verify this rule the following operations are required:

- Fetch the Bypass instruction;

- Check that the control signals of the WBR (Wrapper Boundary of Figure 1) cells are set to work in normal mode.

The static check phase provides information about the wrapper control signals, the code of the bypass instruction and the configuration of the boundary cells needed to understand their operation mode (i.e., if the cells work in normal or test mode). The BFM and the monitor needed to verify the proposed rule are reported in Figure 8 and Figure 9 respectively. Since this rule does not require any timing check, only the scoreboard is implemented.

```
<'
rule_7_4_1_d () @sys.ieee_env.start_7_4_1_d is {

 -- Perform WS_BYPASS instruction fetch
  start sys.ieee_env.ieee_input.instruction_fetch
  (pack(packing.low,sys.ieee_env.ieee_constant.WS_BYPASS));
  emit sys.ieee_env.ieee_input.instruction_fetch_e;
  wait delay (sys.ieee_env.ieee_constant.time_instruction_fetch);

  -- Here all the signals checked in this rule aren't
  -- released because the instruction fetch releases all
  -- the signals that it forces

  for each in sys.ieee_env.ieee_constant.cell_mode do {
   mode_tmp = sys.ieee_env.ieee_output.get_value
              (sys.ieee_env.ieee_constant.cell_mode[index]);
  -- Read the mode signals value
   dut_mode.add(mode_tmp);
  };

  -- Release all the forced design signals
  sys.ieee_env.ieee_input.release_all_signals();
  --  wait delay(1);

  emit check_7_4_1_d;
  dut_mode.clear();

  emit sys.ieee_env.end_7_4_1_d;
 };
};


extend ieee_env_u {

   unit_7_4_1_d: unit_7_4_1_d_u is instance;

   run() is also {
       -- Start TCM implementing rule 7.4.1.d
        start unit_7_4_1_d.rule_7_4_1_d();
   };
};

'>
```

Fig. 8. Rule BFM

```
<'
unit ieee_scoreboard_7_4_1_d {

    check_value() is {

        -- Check that WS_BYPASS set the WBR cells in normal mode
        if (sys.ieee_env.unit_7_4_1_d.dut_mode !=
        sys.ieee_env.ieee_constant.cell_normal) {
                dut_error("Rule 7.4.1.d failed.");
        } else {
                out("Rule 7.4.1.d passed.");
        };
    };
};

extend ieee_output_u {
    scoreboard_7_4_1_d : ieee_scoreboard_7_4_1_d is instance;
};


extend unit_7_4_1_d_u {

    on check_7_4_1_d {
        sys.ieee_env.ieee_output.scoreboard_7_4_1_d.check_value();
    };
};
'>
```

Fig. 9. Rule Monitor

V. Experimental results

This section reports the experimental results obtained by using the proposed verification environment to verify the IEEE std. 1500 compliance of a core implementing a four bit counter with the following characteristics:

- A CLOCK input used as counting clock;
- A RESET input to reset the counting state;
- A LOAD input to force a new start value for the counting;
- A 4 bit input DIN that indicates the start value used when LOAD is high;
- A 4 bit output COUNT that indicates the actual counting value.

This core has been wrapped with a IEEE std. 1500 core test wrapper having the following characteristics:

- An instruction register (WIR), 3 bit length;
- A bypass register (WBY), 1 bit length;
- A boundary register (WBR), 8 bit length;
- An optional TransferDR wrapper serial control;
- Four implemented instructions: WS_BYPASS, WS_PRELOAD, WS_INTEST, WS_EXTEST.

After a complete analysis of the IEEE std. 1500 specification [3] we have been able to identify a set of 165 mandatory rules and 27 optional rules. In our prototype implementation we have been able to deal with the verification of 138 mandatory rules summarized as follows:

- Semantic Rules (they include semantic aspects only): 40;
- Behavioral Rules (they include behavioral aspects only): 25;
- Mixed Rules (they include both semantic and behavioral aspects): 73.

For each behavioral and mixed rule a Specman Elite^{TM}component implementing the rule verifier has been designed according to the guidelines introduced in Section IV-B.

Figure 10 provides an example of the report provided to the user in case of a compliant core. For each rule the verification result (i.e. passed or failed) is reported. Moreover, the user can select the desired rule(s) and display the corresponding simulation waveforms to deeply investigate the signals driven and controlled during the simulation.

At the end of the behavioral verification, the tool provides the coverage measure reached during the simulation (Figure 11). The coverage is useful in order to understand the rules that passed the verification and the level of verification accuracy.

In order to carefully validate the verification capabilities of the proposed environment we designed a set of different Core Test Wrappers, systematically violating different rules of the standard. Figure 12 shows the result of the behavioral verification in case of a non compliant wrapper. In the example, rule number 10.2.1.c fails. The verification environment highlights this violation and also provides the waveform obtained by the simulation to provide a better understanding of the reasons that led to the rule violation. On going work is focusing on creating a violation-programmable wrapper, where different violations can be enabled or disabled in order to verify the efficiency of the verification framework also in presence of multiple violations in the same wrapper.

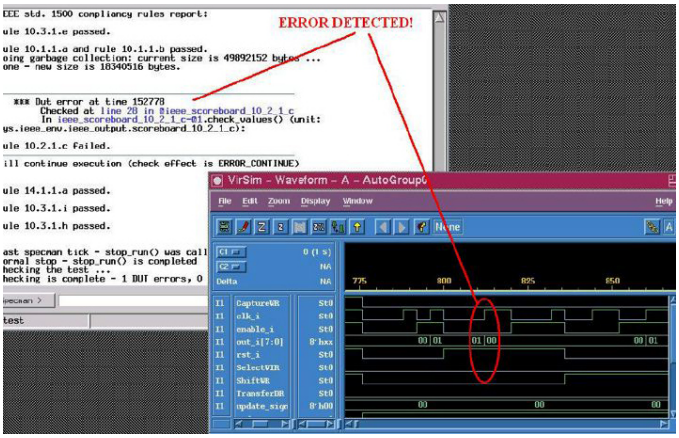

Fig. 12. Error report

VI. Conclusion

In this paper we presented a verification environment for IEEE std. 1500 compliant cores. The environment aims at verifying if the implementation of the Core Test Wrapper provided with an IP core correctly follows the architectural and behavioral rules defined in the standard.

The proposed environment targets different users, from the core designer to the core integrator. It is therefore able to guarantee various level of compliancy depending on the amount of information about the internal core structure available to the user. We believe that in the near future, with the introduction of IEEE std. 1500 compliant wrappers in all IP cores on the market, a verification environment as the one proposed in this paper will be able to increase productivity, reduce design time, and optimize the test plan of very complex SoCs.

References

[1] R. Gupta and Y. Zorian, "Introducing core-based system design," *IEEE Des. Test. Comput.*, vol. 14, no. 2, pp. 15–25, Oct. 1997.

[2] Y. Zorian, "Test requirements for embedded core-based systems and ieee p1500," in *Proc. IEEE International Test Conference, (ITC'97)*. Washington, DC, USA: IEEE Computer Society, Nov.1–6, 1997, pp. 191–199.

[3] *IEEE Standard Testability Method for Embedded Core-based Integrated Circuits*, IEEE Std. 1500, 2005.

[4] Y. Zorian, E. Marinissen, and S. Dey, "Testing embedded-core-based system chips," *IEEE Computer*, vol. 32, no. 6, pp. 52–60, Jun. 1999.

[5] L. Jin-Fu, H. Hsin-Jung, C. Jeng-Bin, S. Chih-Pin, W. Cheng-Wen, S.-I. C. Chuang Cheng, H. Chi-Yi, and L. Hsiao-Ping, "A hierarchical test methodology for systems on chip," *IEEE Micro*, vol. 22, no. 5, pp. 69–81, Sep. 2002.

[6] *IEEE Core Test Language*, IEEE Std. 1450.6, 2005.

[7] T. McLaurin and S. Ghosh, "Etm10 incorporates hardware segment of ieee p1500," *IEEE Des. Test. Comput.*, vol. 19, no. 3, pp. 6–11, May 2002.

[8] S. Picchiotino, M. Diaz-Nava, B. Forest, S. Engels, and R. Wilson, "Platform to validate soc designs and methodologies targeting nanometer cmos technologies," in *Proc. IP-SOC'04*, Grenoble, France, Dec.8–9, 2004, pp. 39–44.

[9] D. Appello, F. Corno, G. M., R. M., and M. Sonza Reorda M.Zorian, "A p1500 compliant bist-based approach to embedded ram diagnosis," in *10th IEEE Asian Test Symposium*, 2001.

[10] I. Diamantidis, T. Oikonomou, and S. Diamantidis, "Towards an ieee p1500 verification infrastructure a comprehensive approach," in *Proc. ClubV*. Grenoble, France: IEEE Computer Society, Mar. 2005.

[11] A. Benso, S. Di Carlo, P. Prinetto, and Y. Zorian, "Ieee std. 1500 compliance verification for embedded cores," *IEEE Trans. VLSI Syst.*, third quarter of 2007.

[12] Uml official website. [Online]. Available: http://www.uml.org/

[13] Specman elite home page. [Online]. Available: http://www.verisity.com/products/specman.html

[14] N. Wirth, *Compiler Construction*. Addison-Wesley, 1996.

[15] Jflex home page. [Online]. Available: http://jflex.de/index.html

[16] Cup home page. [Online]. Available: http://www2.cs.tum.edu/projects/cup/

[17] Vera web site. [Online]. Available: http://www.synopsys.com/products/vera/vera.html

[18] *IEEE Standard e Functional Verification Language*, IEEE Std. 1647, 2006.

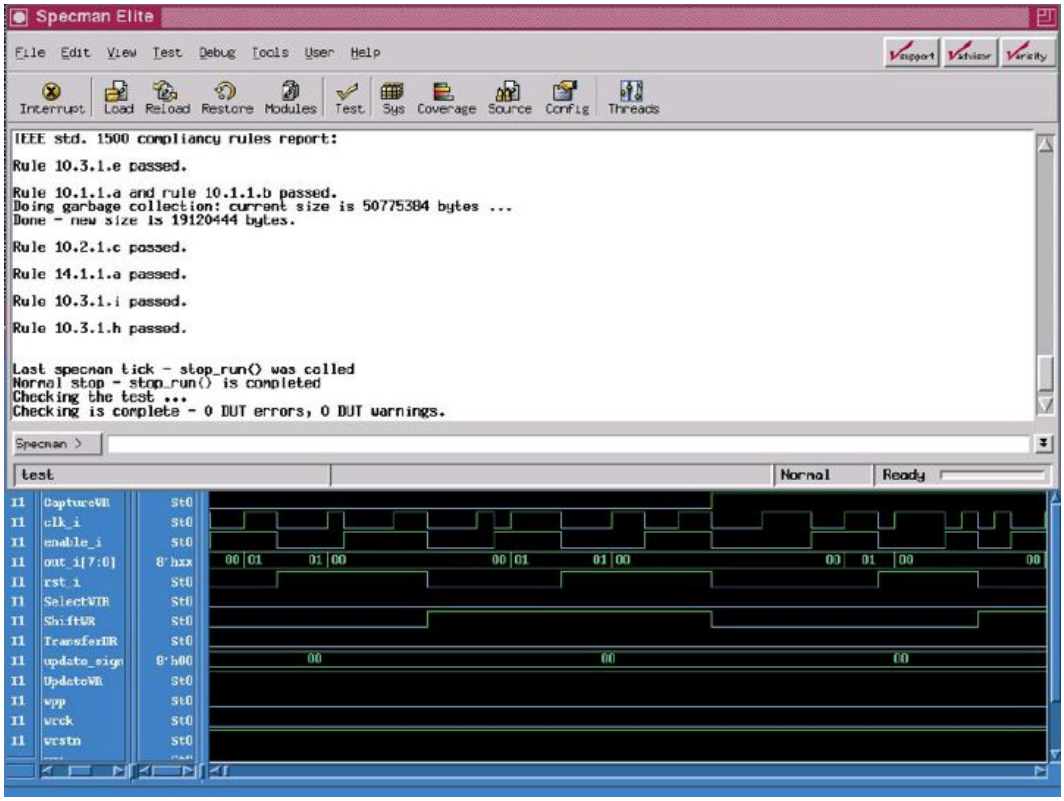

Fig. 10. Compliant wrapper report example

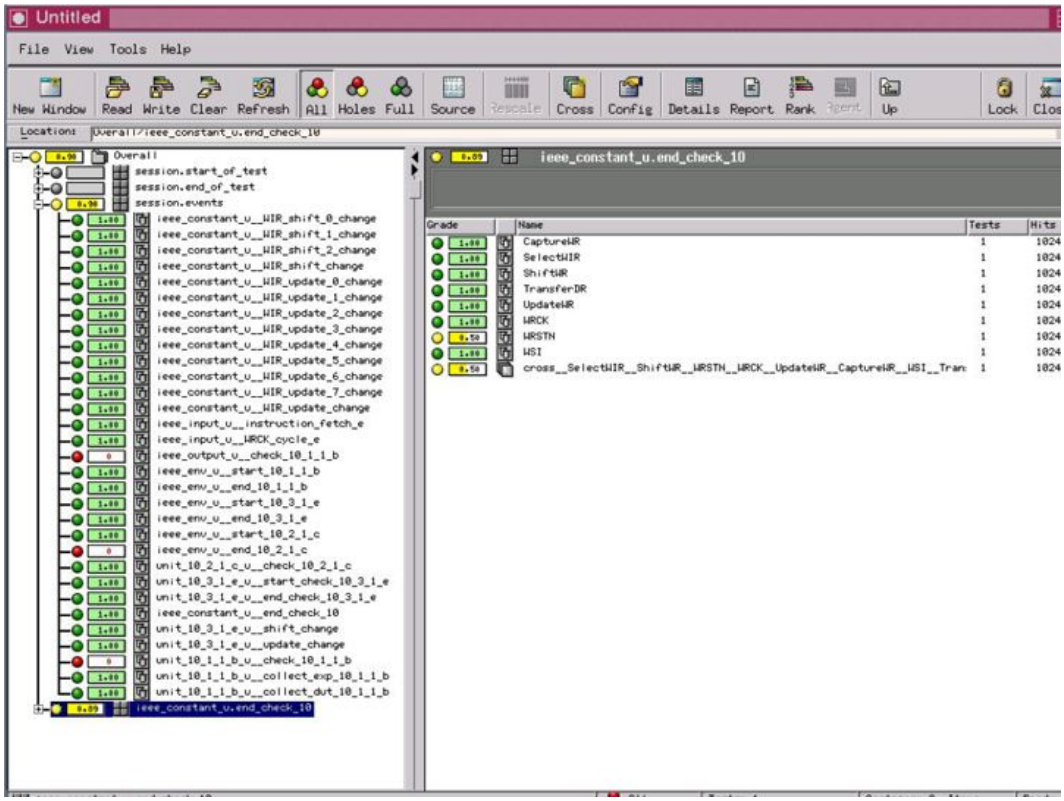

Fig. 11. Coverage report