How to Efficiently Build VHDL Testbenches

Markus Schütz

Siemens AG, Corporate Research and Development, ZFE T SE 5
D-81730 Munich
E-Mail: Markus.Schuetz@zfe.siemens.de

Abstract

The paper describes a reuse methodology, which ease the creation of testbenches. In our approach, beside providing a library of precompiled basic functions and entities, the designer receives descriptions of complete test concepts (e.g. macro-oriented stimulation or comparison of two simulations), including a source code example, called template, and a guide for the adaption of the template to her/his application. Furthermore, an overall guide for the whole validation phase is provided.

The paper describes the typical structure of a testbench, presents the implementations of major objects, and demonstrates the method of user guidance. Further, the global test concept for reuse components is demonstrated.

1 Introduction

The increasing functionality of hardware systems leads to an increasing complexity of the corresponding test cases (stimuli) and simulation results. In practice, the complexity of today's ASIC stimuli and simulation results exceeds the time which can be spent for the simulation phase. Every additional stimulus, which can be simulated and validated before the ASIC vendor's deadline will increase the reliability of the design process. Therefore, **stimuli throughput** improvement is an important objective.

An empirical investigation of [BasPer84] concerning SW design shows, that eight times more errors occur during the development and implementation of single units than during the integration of these units into larger components. Moreover, they show, that in nine of ten cases errors force the modification of a single unit only. Therefore most proposals for software testing techniques strictly recommend a **bottom-up test** approach, i.e. to test single units before test their integration [Bei90,Lig90]. The advantage is an easier error detection and location. Moreover, the description of specific test cases for a single unit is easier in a corresponding unit testbench than in a testbench for a lot of integrated units. Unit test requires single testbenches for these units, which often consist of at least the same number Lines-Of-Code (LoC) as the model itself. I.e. test and implementation of the testbench can require the same amount of effort as for the model itself. Because time is short, HW designers often write only one single testbench for the whole system.

A new kind of test problem comes up with **reuse components** [Prc94]. Often they contain a lot of generic parameters, which control signal levels, signal widths and functional aspects, too. In chapter 6.1 some techniques are briefly presented, how to write testbenches for parameterizable components. The problem of testing the component as completely as possible with one parameter set is a very complex one. The problem of validating all parameter settings can not be solved in this paper (and may be nowhere else), but some ideas are presented in this chapter, which can help the reuser to validate at least her/his parameter set.

1.1 Approach: Testbench Reuse

The concept of reuse is not new. A lot of researchers try to find structured approaches to introduce reuse in the hardware design process [Run94, Prc94]. The main benefits of reuse are:

- use of pre-fabricated modules instead of starting from scratch reduces design time
- use of pre-verified modules reduces test effort.
- reusable components facilitate maintenance, i.e. make it easier for a designer to comprehend and modify a design created by others.

These advantages should be exploited for the testbench design process. The main problem of these approaches is the identification of components or parts of components, which will be highly reused. Reuse is only worth while a component will be often reused. Most of these approaches consider reuse of models for hardware components. However, especially testbenches are good candidates for reuse, because the main structures of testbenches are not as different as those of the components, which will be tested. Testbenches, also for components with absolutely different functionality, consist of a lot of very similar parts. These methods in their global functionality can be nearly independent of the corresponding component, which will be simulated. To take advantage of the main goals of reuse in the testbench design process, the global structure and the main objects of a testbench architecture will be identified first. Afterwards, a set of implemented reusable testbench objects will be presented and their reuse approach inside these testbench architectures will be described.

1.2 Goals and Requirements

The main goal of our approach is the fast and easy creation of testbenches from pre-verified testbench objects. Therefore the time for implementation and test of the corresponding testbench can be extremely reduced and the designer can spend more time for her/his real work, namely the test of the module-under-test.

0-8186-7156-4/95 $4.00

The testbenches itself must be applicable for components of different complexity, i.e. for unit test and, as well for system test. Increasing the simulator's performance is not sufficient to handle large numbers of test cases. The bottleneck concerning stimuli throughput is the effort spent by the designer to describe stimuli and validate simulation results. Rather, techniques must be available, which allow the designer to master the complexity of stimulation and validation. Therefore, s/he has to get techniques to describe stimuli values very compactly and get compacted results of simulation. Abstraction methods for stimulation, techniques for automated validation and protocol functions have to be provided.

As well as the existence of a reusable component alone does not help anybody, the existence of those testbench objects does not, too [Pre94]. Reuse has to be more than a simple library. Somebody has to inform the designers about the existence of components and how to use and adapt them in their testbenches. Therefore, reuser guidance is an important aspect of reuse concepts. The more the designer will be guided, the more efficient the reuse process is. Guiding the reuser by checklists through the whole simulation phase starting from creation of the testbench frame down to gate-level simulation, backannotation and comparison of different simulation runs supported by a library of reusable testbench objects is the result of our approach.

2 Global Test Concepts

Writing VHDL models of hardware components is becoming more and more similar to the software design process. Therefore, some test concepts established in the software design process can be reused in the hardware design process [Bei90, Lig90]:

- **structured bottom up test approach**: start with the test of single units stimulated and validated by single unit testbenches before starting with the test of the component, in which the already tested units are integrated.
- **tester against designer**: if one goes from unit test to component test and up to system test, it is increasingly more effective, if tester and designer are different persons. The more a tester knows about a design, the likelier s/he eliminates useless tests. But the more a tester knows about a design, the likelier s/he has the same misconceptions as the designer. Ignorance of implementation details is the tester's best friend [Bei90].
- **automation**: validation should be as automated as possible. For automatic comparison of output values with expected values, these expected values have to be described by the designer before the simulation starts. This description means additional effort, but more than the act of testing, designing tests (stimuli and expected values) is one of the best bug preventers.

3 Testbench Architecture

The creation of reusable objects is only worth, while they will be highly reused. Therefore we do not want to find one testbench architecture, which fits to all exotic applications, but to the most of them. In this section we identify the main objects of a testbench. In the next section we present several implementations for these identified objects.

Every testbench consists of a **frame**, in which the module-under-test is instantiated. The creation is a very simple task and is already supported by a lot of commercial frontend tools. The second aspect are one or more objects for the **stimulation**. These can be clock generators, simple waveforms, more complex data structures like tables ore files, which will be used for stimuli stream generation or processes and procedures for reactive or very compact algorithmic stimulation. Another important aspect are objects for functional and timing **validation**. Some of the objects used there are very similar or combined with the objects for stimulation. Files or tables can also be used for the description of expected values. Processes or procedures (macros) can be used for algorithmic checks. Automatic validation during simulation should also include objects to generate **protocols**. They inform the designer about the simulation run and result checks in a readable and compact form.

Sometimes additional objects can be necessary. Protocols can also be used to log the stimulation and corresponding reaction of a component or subcomponent in a file, which can be used in another simulation as stimulation and expectation values for this component. The corresponding pendant are objects, which allow the **comparison** of two models, either via the comparison of simulated results with previously protocolled results from an earlier simulation or via common simulation of both models within one testbench.

The next chapter shows implementations of some of these objects.

4 Testbench Objects

Only some testbench object implementations are presented in this chapter to demonstrate the different kinds of effort, which has to be spent by the designer for the adaption of the prepared objects. They are all especially written for reuse, i.e. high reusability as well as fast and easy adaption are very important aspects. An overview of all already existing testbench objects and complete testbenches, their functionality, size and additional features are summarized in Section 7 (Table 1).

4.1 Clock Generation

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY clk IS
  GENERIC (
    ClkStartVal: std_ulogic;-- start value
    ClkDelay   : time ;      -- first edge
    ClkWidth   : time ;      -- second edge
    ClkPeriod  : time ;      -- period
    ClkStop    : time );     -- max simul time
    PORT(ck  : OUT     std_ulogic);
END clk;
ARCHITECTURE ClkArch OF Clk IS
BEGIN
ClkGen : PROCESS
 BEGIN  --  PROCESS ClkGen
  ck <= ClkStartVal;
  Clkloop : WHILE now < ClkStop  LOOP
  ck <= TRANSPORT NOT ClkStartVal AFTER ClkDelay,
        ClkStartVal AFTER ClkWidth+ClkDelay;
  WAIT FOR ClkPeriod;
  END LOOP Clkloop;
  ASSERT false
    REPORT "Clock generation stopped via ClkStop"
      SEVERITY ERROR;
  WAIT;
 END PROCESS ClkGen;
END ClkArch;
```

Figure 1: Reusable Clock Generator Object

The clock generator shown in Figure 1 is a **fully parameterizable model**, i.e. all functional aspects can be controlled via generic constants. In this case the entity can be

precompiled for the designer. For easier adaption the corresponding parameters of the instantiated clock generator are declared as deferred constants inside a package ClkParameter (Figure 2).

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
PACKAGE ClkParameter IS
  CONSTANT testClkStartVal   : std_ulogic;
 ...more declarations
END ClkParameter;
PACKAGE BODY ClkParameter IS
--*********************************************
-- ADAPTION: clocks
-- Specify deferred constants to describe
-- the behavior of your clock signal.
--*********************************************
 CONSTANT testClkStartVal:std_ulogic:= '0';
 CONSTANT testClkDelay  : time      :=  0 ns;
 CONSTANT testClkWidth  : time      := 50 ns;
 CONSTANT testClkPeriod : time      :=100 ns;
 CONSTANT testClkStop   : time     := 4 ms;
END ClkParameter;
```

Figure 2: Package ClkParameter for specification of clock parameters.

Reusing the clock generator is very simple. The clock generator component has to be declared and instantiated inside the testbench (Figure 3) and the corresponding parameters have to be specified inside the package ClkParameter. The source code of the declaration and instantiation can also be given to the designer and directly included into the testbench. Only in the case of multiple clock generator instantiations inside one testbench the names of the parameters of each instantiation have to be changed and included into the parameter package by the designer.

```
.........
USE WORK.ClkParameter.ALL;
.........
 COMPONENT clk
   GENERIC (
     ClkStartVal : std_ulogic;-- start  value
     ClkDelay    : time ;      -- first  edge
     ClkWidth    : time ;      -- second edge
     ClkPeriod   : time ;      -- period
     ClkStop     : time);      -- max sim time
   PORT ( ck : OUT  std_ulogic);
 END COMPONENT;
.........more declarations
BEGIN
  testClk:clk
    GENERIC MAP(
              ClkStartVal => testClkStartVal,
              ClkDelay    => testClkDelay,
              ClkWidth    => testClkWidth,
              ClkPeriod   => testClkPeriod,
              ClkStop     => testClkStop )
    PORT MAP    (Ck       => testClock );
  .......
```

Figure 3: Reuse of clock generator.

4.2 Table-oriented Stimulation and Validation

The following listings show the implementation of a technique, in which stimuli and expected values can be described within a table. Every row corresponds to a stimulation and validation point within a clock cycle. The data structure (RECORD stimuli) for the table is declared within a package stimuliPack (Figure 4).

Constants for stimulation control, i.e start cycle of stimulation startCycle, exact stimulation time point CheckTimeInClk and the repetition of the whole stimulation repeat, as well as the stimuli table stim itself are declared as deferred constants to allow a fast and easy modification by the user (Figure 6). Within each row stimuli and expected values, the number of clock cycles, after which the validation and the next stimulation should take place, the enabling of validation and a logic reference address of the row can be specified. The corresponding process, which has to be placed within the testbench and performs the stimulation and validation, is shown in Figure 5.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
PACKAGE stimuliPack IS
  CONSTANT    repeat  : positive ;
  CONSTANT startCycle    : natural;
  CONSTANT CheckTimeInClk: time ;
  TYPE check_line IS ('N', 'C');
  TYPE   stimulus IS RECORD
--*********************************************
-- ADAPTION: stimuli signals
-- Add a signal declaration for every signal,
-- which should be stimulated with this table.
--*********************************************
     stim1 :    std_ulogic;
     stim2 :    std_ulogic;
--*********************************************
-- ADAPTION: validation signals
-- Add a signal declaration for every signal,
-- which should be validated with this table.
--*********************************************
     expect:    std_ulogic;
     cycles:    positive;
     check : check_line;
     reference: string(1 TO 2);
  END RECORD;
  TYPE stimuli IS ARRAY ( positive RANGE <> )
                  OF stimulus;
  CONSTANT stim: stimuli;  -- deferred.
END stimuliPack ;
```

Figure 4: Declaration of the table stim for synchronous, table-oriented stimulation and validation.

```
USE WORK.stimuliPack.ALL;
USE WORK.ClkParameter.ALL;
ENTITY andTestEntity IS
END andTestEntity ;
ARCHITECTURE stimulation OF andTestEntity IS
.....some declarations
stimul:PROCESS
BEGIN
  WAIT FOR startCycle * ClkPeriod;
  WAIT FOR checkTimeInClk
  repeatStimuli:FOR rep IN 1 TO repeat LOOP
   readTab:FOR i IN stim'left TO stim'right LOOP
--*********************************************
-- ADAPTION: Stimulation
--Assign stimuli values from table stim to the
--corresponding signal,which should be stimulated.
--*********************************************
     Input1 <= stim(i).stim1;
     Input2 <= stim(i).stim2;
     WAIT FOR stim(i).cycles * ClkPeriod;
--*********************************************
-- ADAPTION: Validation
--Add corresponding signal checks for all signals,
--which should be validated with this table.
--*********************************************
     IF (stim(i).check = 'C' ) THEN
       ASSERT( output = stim(i).expect )
         REPORT "wrong value"& stim(i).reference
           SEVERITY NOTE;
     END IF;
   END LOOP readTab;
  END LOOP repeatStimuli;
  WAIT;
END PROCESS stimul;
.....
END stimulation;
```

Figure 5: Process for stimulation and validation

The adaption of this technique cannot be completely controlled via parameters. Therefore a **template approach** is used, i.e the designer gets a source code example. To speed up the adaptions the designer will be guided via checklists. An example of a checklist and their references into the source code are shown in Section 5.

Additional templates and extensions of table-oriented stimulation and validation techniques for other applications are implemented. E.g. a corresponding asynchronous stimulation and validation technique exists, in which discrete time values between two stimulation points can be specified instead of a number of clock cycles. Moreover, templates for synchronous and asynchronous stimulation and validation using TEXTIO files are prepared.

```
PACKAGE BODY stimuliPack IS
--*********************************************
-- ADAPTION: stimulation control parameters
-- Definition of the values of the deferred
-- constants.
--*********************************************
  CONSTANT  repeat         : positive := 200;
  CONSTANT  startCycle     : natural  := 3;
  CONSTANT  CheckTimeInClk : time     := 2 ns;
--*********************************************
-- ADAPTION: stimuli values
-- Describe your stimuli corresponding the row
-- declaration inside the package declaration.
--*********************************************
  CONSANT stim : stimuli :=
  (-- Input1|Input2|Expect|cycles|check|ref
  (     '0',   '0', '0',   1,     'C', " 0"),
  (     '0',   '0', 'H',   3,     'C', " 1")
        );
END stimuliPack ;
```

Figure 6: Specification of stimuli table for table-oriented stimulation and validation.

4.3 Macro-oriented Stimulation

In this section a stimulation method is described, in which stimuli protocols can be described within procedures (macros, see Figure 7). This stimulation method uses stimuli abstraction (macros) and can be used for component as well as for system test.

```
PACKAGE macroPack IS
........procedure declarations
END macroPack;
PACKAGE BODY macroPack IS
PROCEDURE macroRead(
 addressIn : IN std_ulogic_vector(2 DOWNTO 0);
 expDataIn : IN std_ulogic_vector(7 DOWNTO 0);
 SIGNAL address:OUT std_ulogic_vector(2 DOWNTO 0);
 SIGNAL data :IN std_ulogic_vector(7 DOWNTO 0)) IS
 BEGIN
   ASSERT false REPORT "READ Operation started"
     SEVERITY NOTE;
END macroRead;
PROCEDURE macroWrite(
........
END macroWrite;
END macroPack;
```

Figure 7: Implementation of macros inside the package macroPack.

```
PACKAGE programPack IS
 -- ************************************
 -- ADAPTION : data structure
 -- Adapt data structure to macros
 -- ************************************
 TYPE CommandNames IS ( REA, WRI );
 TYPE ProcCall IS RECORD
  command : CommandNames;
  address : std_ulogic_vector(2 DOWNTO 0);
  data    : std_ulogic_vector(7 DOWNTO 0);
  rep     : positive;
 END RECORD;
 TYPE ProcCalls IS ARRAY(positive range <>)
                OF ProcCall;
 CONSTANT repProg : positive;
END programPack;
PACKAGE BODY programPack IS
  CONSTANT repProg  : positive := 1;
  CONSTANT Program : ProcCalls :=
   (-- Command  address  data   repetitions
      ( REA,   "110", "00000000",   1),
      ( WRI,   "000", "11111111",   1));
END programPack;
```

Figure 8: Package programPack, which contains data structure and value of program.

The macros can be called from a program, which is a table (VHDL array of records), in which every row corresponds to a macro call. The table is declared as a deferred constant, which allows a fast and easy modification of the program (Figure 8). Figure 9 shows the corresponding interpreting process, which reads the macro calls from the table and calls the corresponding VHDL procedures described in the package macroPack (Figure 7).

```
macroStim : PROCESS
BEGIN  -- PROCESS macroStim
repeatProgram:FOR rp IN 1 to repProg LOOP
 runProgram  :FOR step IN Program'RANGE LOOP
  repeat:FOR num IN 1 TO Program(step).rep LOOP
   WAIT UNTIL Clock = Clk_sensitivity;
   CASE Program(step).command IS
     WHEN REA  =>  -- call READ procedure
        macroRead(-- data from cmd call
          addressIn     => Program(step).address,
          expectedDataIn => Program(step).data,
          address       => addressbus,
          data          => databusout);
     WHEN WRI  =>  -- call WRITE procedure
        macroWrite(--   data from cmd call
           addressIn     => Program(step).address,
           dataIn        => Program(step).data,
          address        => addressbus);
   END CASE;  --  Program.command
  END LOOP repeat;
 END LOOP runProgram;
END LOOP repeatProgram;
write(protocolLine,NOW,RIGHT,8,ns);
write(protocolLine,STRING'("PROGRAM END"));
writeline(protocolFile,protocolLine);
WAIT;
END PROCESS macroStim;
```

Figure 9: Process for macro-oriented stimulation.

This stimulation technique also uses the template approach. First the designer has to include the process from Figure 9 into her/his testbench and identify the macros, s/he wants to use. For each additional macro s/he adds a corresponding procedure call into the case statement in the interpreting process. Afterwards s/he describes the behavior of all macros inside the package macroPack. Then s/he adapts the progam structure in the package declaration programPack (Figure 8) and starts programming in the package body. A corresponding implementation exists, in which the program can be written inside an ASCII-file.

5 Testbench Creation and Adaption

Step	Action	Ref
2	.....	
3	Choose the implementation form: ○ Data structure: ☐ Table in package body ☐ ASCII files ○ Timing behavior ☐ Synchronous ☐ Asynchronous ○	 p. 10 p. 15 p. 25 p. 30
4	.....	

Checklist 1: Stimulation and validation

Figure 10: Example for the identification of necessary objects and information about them inside a checklist.

The main reason for the acceptance of testbench objects is a fast and easy composition of objects to a complete testbench. The designer must be guided during the whole phase of testbench creation through the identification of necessary objects and information about already existing objects. This is done by well-structured checklists combined with documentation (Figure 10). "Ref." contains a reference to the detailed explanation inside the documentation. Another important aspect is the adaption of the existing templates. The reference from a checklist into the corresponding

source code is done via adaption labels. Every source code part, which has to be adapted, is marked with a special adaption label name:

```
.....
ENTITY TestEntity IS
END TestEntity ;
ARCHITECTURE stimulation OF TestEntity IS
-- ADAPTION: validation                    adaption label
--Add corresponding signal checks for all signals,
--which should be validated with this table.
END stimulation;
```

The checklist contains a reference to the adaption label validation in the corresponding file TestSIT.vhd:

Step	Action	Ref
8	.....	
9	○ Adapt the validation part **File:** TestSIT.vhd **Label:** validation	p. 104
10	.....	

Checklist 2: Table-oriented stimulation and validation

6 Complete Testbenches

The guided creation of a complete testbench out of single testbench objects help designers to improve the testbench design process. The identification of a main testbench structure with single objects leads to a high flexibility, because the designer can decide, which parts will be included into her/his testbench and which implementations will be selected for these objects. On the other hand s/he has to spent some effort for the conception, creation and the test of the testbench. As a very time-consuming task the description of module-under-test-specific test cases remains. Preparation of support for the creation of test cases is very difficult, because they always depend on the module-under-test.

The more details are known about the module-under-test, the more details can be prepared for the corresponding testbench. But, preparation of a complete testbench does only make sense, if the testbench will be highly reused. Frequent reuse is not true for all testbenches, but for some objects of a testbench. Therefore, we decided to identify only the main objects of testbenches and prepared the coreresponding VHDL code. In case of components which are often reused, the so called reuse components, also the testbench is reused often. In this case, the functionality of the module-under-test is known and a lot of very specific testbench code can be prepared including specific test cases. The adaption of a complete testbench will be done much more faster than the creation of a testbench from prepared single testbench objects.

6.1 Testbenches for Reusable Components

Simulation is not only a main step during the development phase of a reuse model. The reuser can expect a pre-tested, but never completely tested, released reuse component in the library. The already existing complexity problem of a complete test of conventional, non-parameterizable models increases with every additional model parameter. Moreover, the template approach of some reuse components includes error-prevention techniques, but cannot ensure an error-free composed model. Therefore, the final validation of the reuser-parameterized or composed model remains a task for the reuser.

Functional Test during Development Phase

An important aspect of the test concept follows proposals of most SW testing techniques [Bei90,Lig90] in which a bottom-up test approach is strictly recommended. First, small base units (unit test) should be tested as completely as possible, before the integration of these units into larger modules and these modules into components will be tested (module and component test). This causes the creation of several testbenches for units, modules and components as well. A complete unit test with all possible parameter settings, as well as a module and system test, was impossible because of complexity problems. Only basic functionalities with some parameter settings could be tested.

The corresponding testbenches must be highly comfortable concerning the adaption of the parameter set, the stimulation and the validation technique. Reuse component parameters are declared as deferred constants in a package body to allow a fast and easy adaption of the testbench. Moreover objects for stimulation techniques, functional checks, timing checks and protocol functions are included in the testbench, which are automatically adapted via the parameters of the reuse component. However, this does not allow the simulation of all parameter settings, but makes it easier to simulate and validate some different parameter settings with nearly automatic adaption of stimulation and validation.

Functional Test during Reuse Phase

For the final test of the fully reuser-parameterized model the testbench already exists from the development phase, fortunately. In the case of a fully parameterizable reuse component stimulation and validation are automatically adapted via the parameters of the reuse component. There is no need for manual adaption of the testbench. Basic testcases are also prepared. Only some reuser-defined testcases could be added, in which the reuser is guided via a checklist. In the case of a reuse component using a template approach the corresponding testbench from the development phase can be used as a testbench template. The reuser will be guided by a checklist during the source code adaption of the template. This also includes the adaption and extension of prepared test cases.

```
-- STEP 2A:WRITE/READ test
--Write all registers with NOT(reset value)
--and read them. If register exist, expect '1',
--otherwise '0'.
(  WRIT,   "000",   "1111111111111111",     1  ),
(  READ,   "000",   "0000000000000000",     1  ),
(  WRIT,   "001",   "1111111111111111",     1  ),
(  READ,   "001",   "0000000000000000",     1  ),
.......
```

Figure 11: Adaption of prepared test cases

Figure 11 shows a part of prepared (macro-oriented) stimuli for a reuse component. The description also includes guidelines for the adaption of test cases in dependence on functional blocks of the module under test, in this case registers.

7 Results

Object Type	*Objects*	*LoC*	*Docu. pages*	*Check-list*	
Stimulation	2	690	4		
Stimulation/ Validation	14	6400	49	1	
Validation	2	700	5		
Protocolling	3	410	6		1
Comparison	4	3700	22	2	
Simulator scripts	4	1180	10		
Backannotation	1	790	10	1	
Other objects	12	1470	27	1	
Complete Testbenches	3	7100	40	3	
Sum	**45**	**22440**	**173**	**9**	

Table 1: Existing testbench objects

Table 1 lists all existing testbench objects separated by their functionality. The table also includes complete testbenches created for some reuse models. Concerning the checklist column there is one global checklist, which guides the designer through the whole phase of simulation. Moreover this checklist contains references to other (sub)checklists. The guided reference to these testbench objects reduces the effort spent for the implementation and test of testbenches and therefore increases the productivity in the hardware design process. The speed up for the creation of testbenches out of these testbench objects compared with a creation starting from scratch is between five and ten.

In practice most designers have their own, but local, library, in which testbenches from previous projects are stored. Therefore, nobody has to start from scratch. But a standardized procedure for the creation of a testbench leads to similar, documented and comprehensible testbench styles and therefore to easier portability of testbenches between different designers. This becomes more and more important with increasing project complexities. Hence, the use of standardized and documented techniques improves the design process quality. Moreover, the use of the objects inside a testbench allows a comfortable stimuli description and a nearly automated validation, which improves the stimuli throughput and therefore directly leads to higher product quality.

Complete testbenches for some reuse components were created using the existing testbench objects. The reuse library contains a 600LoC fully parameterizable error-check component, which is tested via an algorithm within a 400LoC testbench. In this case only the simulation time has to be spent by the reuser for the unit test, because the testbench is automatically adapted via the components parameter setting. For another 2600LoC error-evaluation component, which is also fully parameterizable, a 2700LoC testbench is prepared including 1100 test cases. The reuser is guided via a checklist during the adaption of some optional reuser-defined test cases. Concerning a testbench for a template-based reuse component a 2700LoC interface template is released with a 4000LoC testbench template containing 900 prepared test cases. The adaption of the testbench can be done in about one day by a reuser, which is a very small fraction of the effort used for the creation of a 4000LoC testbench from scratch.

8 Conclusion

The paper presents a methodology for the fast and secure creation of powerful testbenches, which is necessary for a structured bottom-up test approach. The keyword is testbench reuse including a reuser guidance through the whole validation phase. The main objects of each testbench are identified and some exemplary implementations are presented. Moreover, important implementation aspects of user guidance are shown. The set of objects is not restricted to precompiled functions or entities, which can be controlled via parameters. Rather, templates of whole test concepts and a corresponding guide for their adaption are given to the user. This enables higher support to the designer than a library of precompiled, and therefore less flexible objects can do. Furthermore, the problem of parameter test in reusable components is mentioned in the paper and a corresponding test concept is presented. The final validation of a reuse component always has to be done by the reuser. Therefore a testbench will be released together with the reuse component. In this case complete testbenches also including test cases can be prepared and their creation or adaption, respectively, can be guided.

9 Future Works

The templates for a lot of testbench objects already exist. Furthermore, guidelines for their adaption and composition to an application specific testbench are worked out. Hence, it will be only a small step to completely automate the adaption and composition of the testbench objects for a specific module-under-test. This would allow a faster and more secure testbench creation process and leads the designer earlier to her/his real problem, namely the test of her/his module. Moreover, further testbench objects will be identified and implemented.

Acknowledgements

I would like to thank Sabine Roessel for her useful suggestions and Dieter Zerweck for supporting the application in several ASIC projects.

References

[BasPer84] Basili V.R, Perricone B.T, *Software Errors and Complexity: An Empirical Investigation*, Communications of the ACM, Vol 27, No.1 January 1984, pp. 42-52

[Bei90] Beizer B., *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1990

[Lig90] Liggesmeyer, Peter: *Modultest und Modulverifikation*, BI-Wissenschafts-Verlag, 1990, Mannheim, Germany

[LRM] *IEEE Standard Language Reference Manual*, Institute of Electrical and Electronisces Engineers, 1988

[Pre94] Preis V., *An Approach to Complex and Self-Generating VHDL Models for Simulation and Synthesis*, VHDL-FORUM for CAD in EUROPE, 1994, Italy

[Run94] Runner J. Scott, *Considerations When Implementing a Design Reuse Methodology, Synopsys Methodology Notes*, March 1994