Semi-Formal Test Generation with Genevieve

Julia Dushina
STMicroelectronics
1000 Aztec West,
Bristol BS32 4SQ, UK
dushinaj@bristol.st.com

Mike Benjamin
STMicroelectronics
1000 Aztec West
Bristol BS32 4SQ, UK
benjamin@bristol.st.com

Daniel Geist
IBM Corp
MATAM, Haifa
ISRAEL
geist@il.ibm.com

ABSTRACT

This paper describes the first application of the Genevieve test generation methodology. The Genevieve approach uses semi-formal techniques derived from "model-checking" to generate test suites for specific behaviours of the design under test. An "interesting" behaviour is claimed to be unreachable. If a path from an initial state to the state of interest does exist, a counter-example is generated. The sequence of states specifies a test for the desired behaviour.

To highlight real problems that could appear during test generation, we chose the Store Data Unit (SDU) of the ST100, a new high performance digital signal processor (DSP) developed by STMicroelectronics. This unit is specifically selected because of the following key issues:

1. big data structures that can not be directly modelled without state explosion,
2. complex control logic that would require an excessive number of tests to exercise exhaustively,
3. a design where it is difficult to determine how to drive the complete system to ensure a given behaviour in the unit under test.

The Genvieve methodology allowed us to define a coverage model specifically devoted to covering corner cases of the design. Hence the generated test suite achieved very efficient coverage of corner cases, and checked not only functional correctness but also whether the implementation matched design intent. As a result the Genevieve tests discovered some subtle performance bugs which would otherwise be very difficult to find.

1. GENEVIEVE VERIFICATION SUITE

This section briefly describes the Genevieve verification suite (see [1]), a semi-formal test generation tool.

Semi-formal test generation ([2]) has developed from the use of "model-checking" ([3]) to generate test suites for specific behaviour of the design under test. An "interesting" behaviour is claimed to be never reachable while supplying a property to a model-checker. If a path from initial state to the state under interest does exist, a counter-example is generated by a model-checker. The sequence of passed states forms desired test suite to achieve the goal behaviour.

An "interesting" design state is often referenced as **corner case**, which is a composition of border behaviours for different design parts or blocks. In this documents we often use "corner cases" to specify a particular design state we want to test.

Hereafter we describe used concept and terminology and give more detail concerning the methodology itself.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

1.1 Concept and Terminology

While writing tests, it is essential to measure the quality of generated tests. The choice of measurement or **coverage model** depends on the resources available for test generation and simulation. One of the possibilities, available with commercial tools like VHDLCover, is to define the coverage model in terms of lines or statements of a design description. In our example the designers used VHDL as hardware description language. Good tests will cover each VHDL line/statement. The separate elements that compose a coverage model (lines/statement of VHDL source code) are called **coverage tasks**. Thus, a coverage model is defined more formally as a set of coverage tasks.

Other coverage models derived from hardware languages include:

- Branch Coverage consists of all branches in VHDL source code;
- Basic Sub-Condition Coverage consists of all branches in VHDL source code together with the conditions that were met in order to take the branch;
- Path Coverage consists of all possible combination of the sets of branches.

Though language based coverage models are comprehensive and easy to define, they do not reflect sequential behaviour of the circuit and thus real corner cases the designers wish to verify. To cope with this problem, other coverage models have to be chosen relevant to sequence of the design states. The coverage model based on a Finite State Machine (FSM) representation works well for this task.

Moreover, the employment of a formal model-checking tools makes the use of the FSM model natural for both test generation and test quality measurement. So, we will compare the tests obtained with Genevieve and with ordinary verification routines using **state coverage model** briefly described bellow.

A **finite state machine** is defined to be a finite **set of states** and a set of **transitions** from state to state that occur on **input symbols** taken from some finite **environment set**. For each input symbol there is at most one transition out of each state (possibly back to the state itself). Note that not all input symbols are valid in all states. This means that the transition relation may be thought of as a function from a subset of the Cartesian product of the state set and environment set to the state set. A subset of the states are designated to be the **initial states** of the machine, and another subset of the states are designated as the **final states**. Often the following notational conventions are used:

A finite state machine $M = (V, \Sigma, \delta, S, F)$ has state set V, input symbols Σ, transition function $\delta : V \times \Sigma \to V$, initial (or start) states $S \subseteq V$, and final states $F \subseteq V$. An input symbol s is

said to be **valid** at state v if the transition function is defined at (v, s).

A **state coverage model** in this context might be equivalent to the state set V. It means that we are interested in testing every possible state of an FSM. The test suites will cover each coverage task (each state) of the state coverage model if the tests start with one of initial states, finish in one of final states and go through each states of the FSM at least once.

In practice the state set V is derived from some hardware description language (VHDL in our case) and represented as the Cartesian product of the domain sets of **state variables** found in hardware description source: $V = VAR1 \times VAR2 \times \ldots \times VARn$ where $VAR1$, $VAR2$, ..., $VARn$ are the domain sets (possible values) for the state variables v_1, v_2, ..., v_n. Normally, v_1, v_2, ... v_n correspond to signal of variable/register declarations in circuit description.

Typically, designers are interested in testing not all but particular state variables, particular values of state variables, or particular combinations of state variable values. Exclusion of values/variables that are not relevant from the user's point of view makes the coverage model smaller and results in more efficient and more quickly obtained tests. The Genevieve verification tools allow to distinguish between "interesting" and "not interesting" state sets by partitioning of the FSM state variables into three groups:

1. **Coverage variables** are state variables whose values are included in coverage model; designers are interested to tests all values of these variables;
2. **Ignore variables** are state variables whose values are not included in coverage model; these variables are not relevant to test process;
3. **Care variables** are state variables where some values are included in coverage model; designers are interested to test not all but some particular values of these variables.

The partitioning of variables into three groups is made by the user and based on test requirements. Special instructions are conceived for this purpose and discussed later.

Often systems operate under the assumption of some predefined environment. The behaviour of the environment, and therefore the inputs of FSM, depends on reaction of the design under consideration. It is then necessary to model the environment as one or more FSMs that provide legal inputs to the design. The environment behaviour is integrated into the design model, thus extending the state set of corresponding FSM and eliminating its input set. This situation is similar to that during testing with testbenches. The top level testbench circuit contains no inputs. If every variable, signal or register is considered as a state variable, corresponding to the outmost testbench, the FSM contains no inputs but only state set.

The environment becomes part of the input to test generator but we do not wish our coverage models to be dependent on the environment FSMs, because they only represent the legal external stimulus to the design. Fortunately, the variable partitioning helps to separate between design and environment state sets. Thus, most of the design state variables are coverage or case and define coverage model, whereas the environment state variables are ignore ones and do not influence the coverage model.

1.2 Methodology Description

As we mentioned previously, the Genevieve methodology relies on formal methods for test generation. However, we can not apply formal approaches directly due to the complexity of modern designs and related state explosion problems (Figure 1).

To cope with state explosion difficulty, we describe the design under test (DUT) in a simplified manner. This process, called **abstraction,** is shown in Figure 2. While there exist different kinds of abstract mechanism (see [4]), in this work we are concerned mainly with three of them:

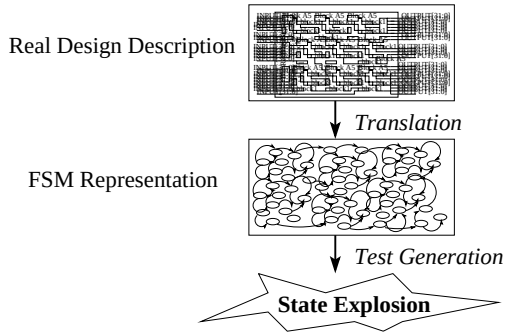

Figure 1. State explosion using conventional formal methods

1. functional abstraction to reveal the main functionality of the design and to hide cumbersome details; the purpose of the testing becomes clear;
2. data abstraction is related to functional one; irrelevant data are grouped into classes or not considered at all;
3. temporal abstraction is interested in the order of events, rather than in precise timing.

Ideally, the abstract description is the same as a (formal) specification for current circuit implementation. Advantages and limitations of abstraction mechanisms are discussed in more detail while describing the tested SDU block.

We use the Mμ ALT (Modelling micro-Architecture Language for Traversal) language for abstract description of the design under test. Mμ ALT is a VHDL based language with the usual VHDL facilities. In addition, it is possible to define test coverage models and test constraints for test generation process. The coverage model is basically determined by adding special attributes to "interesting" signals or variables which are referenced as **coverage variable**s. Thus, the combinations of all possible values of coverage variables constitute the first rough set of interesting corner cases or **coverage model**. Each combination corresponds to a state when the abstract description is translated to a FSM model. Later in this document we use the term "state" to refer a combination of variable values and we say that coverage model consists of coverage states.

The coverage model can be further refined by the means of special functions. Thus, the designer might be interested to test the circuit only with some specific values of some variables or signals.

The test constraints restrict the way targeted coverage states are reached. First of all, initial and final state of the test sequence can be defined. Some states or transitions can be forbidden to appear in the test sequence. It is also possible to put some state between another two states in a test suite.

Finally, Mμ ALT allows non-deterministic expressions. It is especially useful for input assignments: the designer can assign a set of values to a signal or variable. One of the values will be randomly chosen during test generation. Some other facilities, like the possibility to define the test length or the number of tests required for each coverage task, are also provided by the Mμ ALT special constructions.

When the abstract description is ready, it is translated to a state machine representation usable by the GOTCHA test generation tool (Figure 2). The intended coverage model is also extracted during this translation from supplementary Mμ ALT constructions. GOTCHA (Generator of Test Cases for Hardware Architecture) is a prototype coverage driven test generator, written expressly for the Genevieve project (see [5]).

The GOTCHA compiler builds a C++ file containing both the test generation algorithm and the embodiment of the finite state machine. The state machine is explored via a depth or breadth first search from each of the start states. Progress reports on this initial state space exploration can be customized in a limited way.

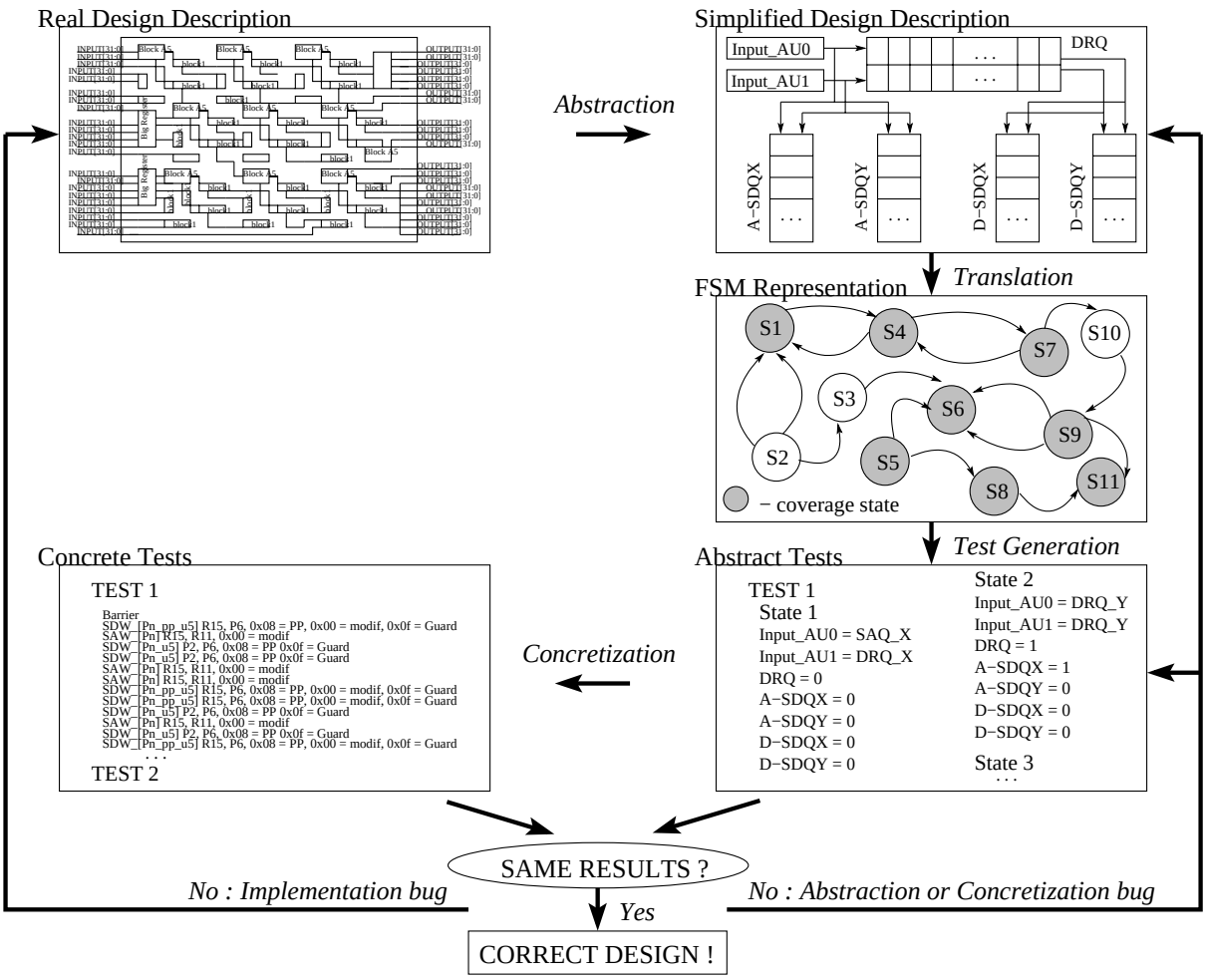

Figure 2. Genevieve test generation methodology

On completion of the enumeration of the entire reachable state space, a random coverage task is chosen from amongst those that have not yet been covered or proved to be uncoverable. A test is generated by constructing an execution path to the coverage task (state in our case) then continuing on to a final state. If the test length recommendation has been exceeded at this point, then the test is output, otherwise an extension path to a further final state is sought, and appended to the test. This process continues until either the test length recommendation is exceeded or final state reached has no path to a further final state. If the randomly chosen coverage task cannot reach a final state then no test is generated.

Thus, the GOTCHA tool results in a set of **abstract tests**, each abstract test containing a sequence of states. Figure 2 roughly shows this process. A state is determined by concrete values of all state variables (coverage or not). In the Figure 2 the state variables are Input_AU0, Input_AU1, DRQ, A-SDQX, A-SDQY, D-SDQX, and D-SDQY. Design variables contain both state variables in a proper sense (it means variables or signals that represent real design's registers) as well as input variables. Thus, in Figure 2 the variables DRQ, A-SDQX, A-SDQY, D-SDQX, and D-SDQY represent proper states of the design under test, whereas the variables Input_AU0 and Input_AU1 represent the design's inputs.

Abstract tests give sequence of states to reach coverage task. They can not be directly applied to the real design. In order to obtain real or **concrete tests**, we have to make **concretization** of abstract tests. The concretization consists of two major transformations. Firstly, the design variables not corresponding to the design inputs are removed from each state of abstract test. After this operation an abstract test sequence contains abstract inputs only. Then, the remaining input variables are replaced with the inputs of the concrete test. The input of the concrete test has to provide intended values to real design inputs. Normally, every value of each abstract input variable demands separate concrete counterpart.

The level and structure of concrete test inputs depend on test objectives. It may be just supplying values to the design inputs via simulator commands or microcontroller instructions if the design is tested at functional level. In addition, a preamble and epilogue test suites are almost always required in order to reset the real design before test and compare the results after test. Normally the concretization is done during straightforward translation. At the end of concretization process real test suites are ready for simulation or emulation.

After simulation/emulation of the real design, obtained real test results have to be compared with expected abstract ones. The comparison can be seen as a process opposite to the concretization: abstract test results are represented by the design state variables in a proper sense. It means that input variables (Input_AU0 and Input_AU1) have to be removed from abstract tests and then remaining variables compared to real test results. As abstract and real design description can differ considerably, the relation between abstract and real state variables must be established. In the example of Figure 2, we have to find and display signals/variables of the real design that correspond to the abstract variables DRQ, A-SDQX, A-SDQY, D-SDQX, and D-SDQY.

The comparison itself is done for each state of abstract test. In general, the comparison is successful if every abstract state variable has the same value as corresponding signals/variables of the real design. It is however possible that not all abstract state variables need to be compared or a matching function is required for comparison.

If temporal abstraction was not used during abstract design description, then successive states of abstract test must correspond to successive states of the real design (cycle accurate). Otherwise, supplementary states (clock cycles) may exist between real design states that match abstract design states. We say in this case that abstract and real design descriptions have different time scale.

If the results of abstract and concrete tests match, then the design implementation is correct and satisfies intended behaviour expressed by the abstract description. If not, then three scenarios are possible. First, the implementation is not correct and has to be modified. Second, the abstraction is wrong: the design's functionality is misunderstood or too much details are omitted. Third, the concretization does not supply intended abstract inputs to the real design. In each case a feedback to the problem source is necessary and the whole process has to be repeated.

2. VERIFIED SDU BLOCK

This section is devoted to the description of the SDU block that was chosen as first application example of the Genvieve project. SDU is a part of the ST100, a new high performance digital signal processor (DSP) developed by STMicroelectronics. The verified unit is a block of the Data Memory Controller (DMC) which is responsible for storing data to memory.

The SDU unit has been specifically selected to highlight key issues that the Genevieve project must address:

1. big data structures that can not be directly modelled without state explosion,
2. complex control logic that would require an excessive number of tests to exercise exhaustively,
3. a design where it is difficult to determine how to drive the complete system to ensure a given behaviour in the unit under test.

The SDU block is shown in Figure 3. It inputs data from the ST100 address (AU) and data (DU) execution units which provide respectively address and corresponding data. To achieve a high performance, both the AU and DU blocks are split into two identical subunits (AU0, AU1, and DU0, DU1 respectively) capable of providing two addresses and two data values per machine cycle. While the AU execution unit basically supplies the address for memory stores, it occasionally can supply the data itself. The data can come from the AU unit when, for example, an address pointer register of the AU unit needs to be stored in the memory.

The data from AU and/or DU execution units is routed to Store Data Queues (SDQs) of the SDU and then further to the memory. As the memory is organized into two banks X and Y, the data is held in four separate queues according to both the source and destination (A-SDQx, A-SDQy, D-SDQx, D-SDQy).

This organization requires a routing mechanism to allow stored data to go to the correct bank. When an address is output from the AU-pipe it specifies where the data should be directed (X/Y bank):

- When the data comes from the AU it is directly routed to the correct A-SDQ[xy].
- When the data comes from DU the routing information is stored in a DU Routing Queue (DRQ). The DRQ is a FIFO which records 4 bits of data on each cycle where the AU provides an address for a store from the DU. The information is coded by an X and Y enable bit for each slot. As soon as the corresponding data is available at the output of the DU it is routed to the correct D-SDQ[xy] according to the DRQ directives. If only one slot provides the DMC with an address the 2 bits of the other slot are set to "no store". This makes it possible to preserve the ordering of slot0 versus slot1 when reading DU data. This is necessary because the DU slots can be granted independently.

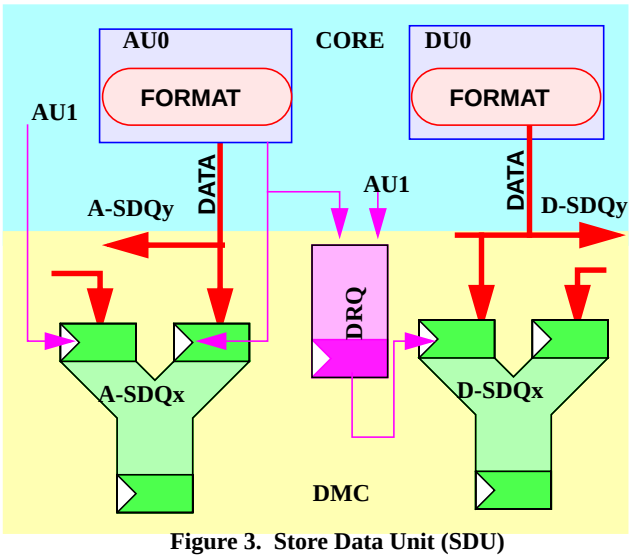

Figure 3. Store Data Unit (SDU)

The SDU unit outputs the data to the X bank memory either from A-SDQx or D-SDQx queues according to the arrival order. The same principle is applied to the data storing in the Y-bank memory.

Figure 3 shows the routing mechanism from the first slots of each execution unit (AU0 and DU0) to the X memory bank only. In the same manner the data are routed from the AU0 and DU0 slots to the Y memory bank and from the AU1 and DU1 slots to the X and Y memory banks. The depth of A-SDQx, A-SDQy, D-SDQx, and D-SDQy queues is nine and the depth of DRQ queue is thirteen.

3. TEST GENERATION FOR SDU UNIT

This section describes how Genevieve test generation methodology was applied to the SDU unit.

The interesting corner cases for this block are those reflecting "border" filling of five principle SDU queues. Each queue is considered to be empty, partially filled (we say valid) or full. For the D-SDQ queues, it is also important to consider when queue is "almost full" (we say quasifull), meaning that one place is left empty in the queue. We refer to the empty, valid, quasifull and full status of a queue as **abstract state** of queue, state of **abstract queue**, or **abstract queue state**.

All possible combinations of abstract queue states constitute the corner cases to test. The corner cases number is then equal to 3*3*3*4*4=432. The test generation objective is to cover as many as possible of these corner cases, taking into account that not all of them are in principle reachable.

3.1 Abstract Model

The abstract model has to capture the essential behaviour of the SDU block, thus concentrating on modeling the five principal queues of the unit. Each abstract queue is represented in the model by a signal that can take "empty", "valid", "quasifull" or "full" abstract value. Unfortunately, the use of abstract queues only is not sufficient for real test suites generation. That's why each abstract queue signal is doubled by a "real" queue signal calculating concrete number of elements in the queue each clock cycle. This is schematically shown in Figure 4.

The number of elements in each real queue is determined based on the present number of elements, coming inputs and whether an element is output to the memory. For example, if the A-SDQx queue contains five elements, two datas arrive from the AU unit both going to the X-bank memory, and one element is output from the queue into the memory, then the number of elements during next

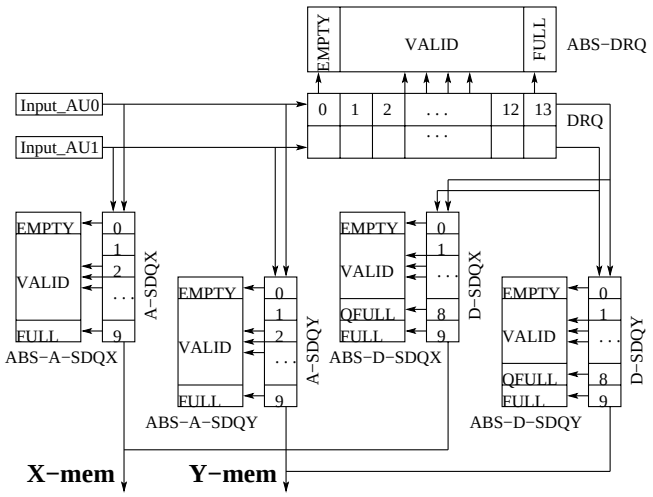

Figure 4. Abstract SDU model

clock cycle is equal to six (6 = 5+2-1). When the real elements number is calculated, a special function maps this real number to an abstract queue state. Thus, for A-SDQ queues zero corresponds to the "empty" abstract state, any number from one to eight corresponds to the "valid" abstract state, and nine corresponds to the "full" abstract state. The output to each memory bank (X/Y) is regulated by a special process that keeps the order of data arrival (from AU or DU execution unit).

As the data can arrive only from one execution unit (AU or DU) at a time, the abstraction is done for the inputs of the SDU block. The four inputs to the SDU block are grouped into two classes: input from slot 0 (Input_AU0) and input from slot 1 (Input_AU1). Each class can be one of the following values designating both the data source and memory destination: "saq_x", "saq_y", "drq_x", "drq_y", and "no_au".

Temporal abstraction is not done for the verified unit. In order to trace generated test suites and to check the expected performance, the numbers of queue elements must have cycle-by-cycle matching in the real design and abstract model. As the sequence of events we are interested in (number of elements in the queues) is cycle accurate, the abstract model has to follow the behaviour of the real design, and not "skip" real design states.

3.2 Abstract Test Generation with GOTCHA

The SDU abstract model was supplied to the GOTCHA tool for abstract test generation. The coverage model is defined by the five signals corresponding to abstract queues (the signals ABS_DRQ, ABS_SAQX, ABS_SAQY, ABS_SDQX, and ABS_SDQY have the special CVAR_SCALAR attribute), thus giving 432 possible coverage states. No refinements of coverage model, like reducing the set of coverage states by the means of special functions or defining transitions to cover, were used. There was also no need to define final states as any state of the model was a valid state to terminate the tests during simulation.

The input signals participate in the global state space definition (section 1.1) and are assigned inside the abstract model. The easiest way to determine the model's inputs is to use non-deterministic assignments: the tool randomly chooses one of the assigned values and then explores all reachable states in order to find a coverage one. If a coverage state is found, an abstract test (sequence of states to the coverage state) is generated.

Unfortunately, the state space of the abstract model is still huge due to non-negligible depth of "real" SDU queues. If the input assignments are completely random, most of the coverage states will never be found because of the state explosion problem. That's why we chose to "guide" the tool towards possible interesting corner cases. This is done by splitting input assignments into several modes. An objective of an assignment mode is to fill or empty certain queues. Thus, we may gradually fill the D-SAQx queue in the first mode and the D-SDQx queue in the second. Then, during test generation the coverage states with full D-SAQx or D-SDQx queues are likely to be found.

Normally, the inputs are guided to cover certain coverage states. Due to the complexity of the design it is not possible to cover all desired coverage states within one model even with guided inputs. The solution we found is to use several versions of the same abstract model, each version guiding inputs to cover a different coverage subset.

Thus, during test generation with GOTCHA we wrote several versions of the same abstract model, each differing in input assignments. These then generated test suites that covered subsets of coverage states. To define the overall coverage with all generated test suites, a simple analysis program was written. This program analyses newly generated tests and adds newly covered states to previously covered state set. It also completes final test suites with new tests. After 24 abstract models no more corner cases were covered so we stopped the test generation process.

Using GOTCHA tool we were able to cover 293 of 432 coverage states. We estimate that a significant part of the uncovered states are not reachable.

3.3 Concretization of Abstract Tests

The target of the test generation process is to obtain tests at a functional level meaning sequences of ST100 instructions. For that, each abstract test generated with GOTCHA has to be translated into corresponding instruction pattern. The principle of concretization is described in section 1.2. The concretization process for the SDU unit is shown in Figure 5.

Let's consider Figure 5 in more detail. The transformation is done in two steps. The first step is translation of abstract test into test specification for Genesys, a model-based test generator. In every abstract test each assignment of the SDU input variables (AU0 or

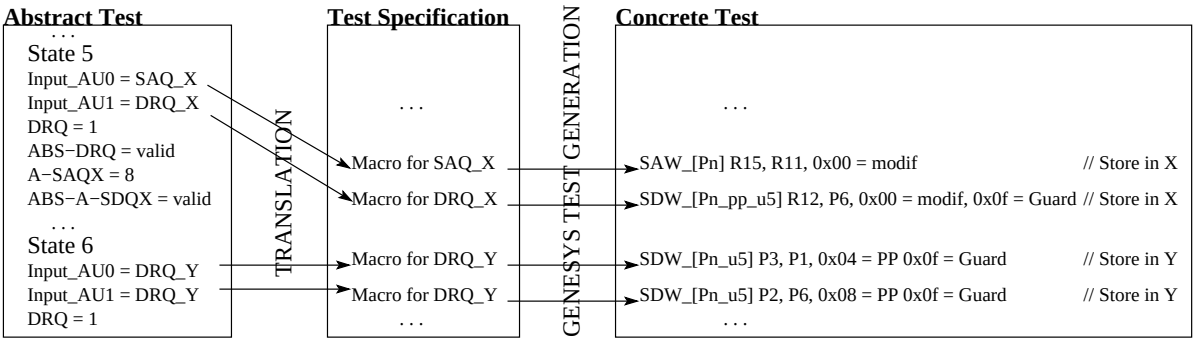

Figure 5. Concretization process for the SDU unit

AU1) is replaced with a Genesys macro allowing to supply desired value to the SDU real inputs. For example, the assignment to the SAQ_X value means that data arrives from the Address Unit (AU) block and goes to the X bank memory. For this particular value a special macro is manually created that:

- uses SAW (Store Address Word) command; the data thus flows from the AU unit because one of its registers needs to be stored;
- defines the store address to be in the X memory bank.

For the DRQ_X/Y values, the SDW (Store Data Word) command is used, meaning that data comes from the Data Unit (DU) block and that routing information is stored in the DRQ queue first.

The sequence of macros corresponding to the inputs from one abstract test constitutes the core for one Genesys test specification. Some reset commands are added in the beginning of each test specification.

The second step is final test generation using Genesys. Each test specification is transformed into instruction sequence ready for simulation. The macros of instruction sequence are translated by Genesys into concrete ST100 commands. The concrete command corresponding to one particular macro can slightly vary from one translation to another. For example, instructions corresponding to the SAQ_X macro can use different registers of the AU block and different store addresses remaining nevertheless in the X-bank memory space.

As the SDU inputs are supplied at functional level by means of ST100 instructions, a modification of intended input sequence is possible. The flexibility is fixed during macro definition.

3.4 Abstract and concrete test results

When concrete tests are generated, they are used for RTL-level simulation of ST100. In order to compare the expected abstract results with concrete ones, the SDU functionality has to be traced during simulation. Special simulator commands are used to record the values of interesting microarchitectural signals.

First of all, we need to check whether intended abstract values are indeed supplied to the SDU inputs. Therefore we display some request/grant signals and the actual SDU input values. Further, to verify the functionality of the SDU block, we record the signals representing the number of elements in each queue and queues status (empty, valid, quasifull or full). The time information (clock cycle number) is also displayed to distinguish states of the real design: each state corresponds to separate clock cycle.

The comparison itself is done by a special purpose program: for each state of abstract test the signal values are checked against corresponding signal values of simulation results. Although the coverage model is defined by signals of abstract queues only, for each queue we compare both the abstract queue status and actual number of elements in the queue. This is done to facilitate the analysis of testing results.

As mentioned before, the temporal abstraction is not used for SDU abstract model. It means that consequent states of abstract test must correspond to consequent states (clock cycles) of the real design.

4. CONCLUSION

4.1 Results

This work resulted in efficient test generation methodology demonstrated on a complex design. We established different steps of the test generation process and finally obtained concrete tests applicable for real design simulation. We also created suitable design verification environment consisting of several translation and comparison programs.

The SDU unit chosen for this experiment has sophisticated behaviour and complex interfaces with other system blocks. The extremely simplified abstract model of the SDU device has 60480000 states (10*10*10*10*14=140000 real queue states multiplied by 432 abstract queue states). We could only generate tests by defining much more smaller coverage model and by guiding inputs in order to reach coverage states.

The possibility to define coverage models is a very important feature of the test generation process. It allows to clearly identify the purpose of testing and to measure the quality of generated tests. It has to be pointed out that before Genevieve approach the designers and verification team mostly used metrics based on covered lines of circuit's hardware description. In practice it appears that these code-based metrics are very weak and do not cover "border" circuit behaviour.

Based on the coverage model we measured the quality of generated tests. Using the Genevieve methodology we managed to obtain tests for 293 of 432 possible corner cases of interest. During comparison step we discovered that not all abstract tests matched simulation results. Some SDU behaviour, while still functionally correct, did not match the original specification. In particular the implementation did not use the whole capacity of the queues as these were never filled if only one place was available in a queue. So, Genevieve tests discovered some subtle performance bugs that would otherwise be very difficult to find.

Due to the mismatching between implementation and specification, the number of corner cases covered by concrete Genevieve tests are less than the number of corner cases covered by corresponding abstract tests. But even so, the tests generated by Genevieve tools still show better results than the tests generated manually by a verification engineer where less corner cases are covered by a bigger number of manual tests. These results are summarized in the table below.

Table 1. Comparison between Genevieve and manual tests

	Abstract	Concrete	Manual
Number of tests	293	293	365
Covered corner cases	293	73	53

Another result achieved with the experiment is the possibility to model complex design environment and to generate tests for a particular system block at a high functional level. We were able to supply intended values to the SDU inputs via sequences of ST100 instructions. This was only possible because the Genevieve verification tools allow modelling the environment of a design during test generation. None of the existing test generation or formal verification tool is able to provide the same facility.

4.2 Future work

Despite some difficulties the Genevieve methodology proved successful and is now being applied to test generation for the DMA controller of ST50, a new general purpose RISC microprocessor developed by STMicroelectronics and Hitachi. Like other memory or memory-related device, the DMA controller challenges formal techniques due to the state explosion problem.

The developing of completely automatic translation tools and well-established verification environment is also envisaged.

5. References

[1] M.Benjamin and all. *A Study in Coverage-Driven Test Generation*, in DAC 99

[2] A. Aharon and all. *Test program generation for functional verification of PowerPC processors in IBM*, in DAC 95

[3] K. L. McMillan *Symbolic Model Checking* Kluwer Academic Press, Norwell, MA, 1993

[4] T. Melham *Abstraction Mechanisms for Hardware Verification* in VLSI Specification, Verification and Synthesis, Kluwer Academic Publisheres, January 1987

[5] D. Geist and all. *Coverage-Directed Test Generation Using Symbolic Techniques*, in FMCAD 96, November 1996