

Semi-Formal Test Generation for a Block of Industrial DSP

Julia Dushina*, Mike Benjamin*, Daniel Geist**

* STMicroelectronics, 1000 Aztec West, Bristol BS32 4SQ, UK, {dushinaj,benjamin}@bristol.st.com

** IBM Haifa Research Lab, MATAM, Haifa, ISRAEL, geist@il.ibm.com

Abstract

This article describes an industrial application of the Genevieve test generation methodology. The Genevieve approach [1] uses formal techniques to generate test suites for specific design behaviour. The example, which is a part of the ST100 DSP, was chosen in order to highlight real life problems such as big data structures, complex control logic, and complex environments where it is difficult to determine how to drive the complete system to ensure a given behaviour in the unit under test.

1. Genevieve Methodology

Semi-formal test generation has developed from the use of “model-checking” ([2]) to generate test suites for specific behaviours of the design under test. An “interesting” behaviour is claimed to be unreachable while supplying a property to a model-checker. If a path from initial state to the state of interest does exist a counter-example is generated by a model-checker. The sequence of states specifies a test to achieve the required behaviour.

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

To cope with the state explosion problem, we describe the design under test (DUT) in a simplified manner. This process, called **abstraction**, is shown in Figure 1. While there exist different kinds of abstract mechanism (see [3]), in this work we are concerned with three of them:

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 abstraction; data is 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 of the current circuit implementation. Advantages and limitations of abstraction mechanisms are discussed in more detail while describing the tested SDU block.

We use the M μ ALT (Modelling micro-Architecture Language for Traversal) language for abstract descriptions of the design under test (see [4]). M μ ALT is a VHDL based language with the usual VHDL facilities. In addition, it is possible to define test coverage models and constraints for test generation. The coverage model is basically deter-

mined by adding special attributes to “interesting” signals or variables which are referenced as **coverage variables**. 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 an FSM model. Later in this document we use the term “state” to refer to a combination of variable values and we say that a coverage model consists of coverage states.

The coverage model can be further refined by means of special functions to only test specific values of some variables or signals.

The test constraints restrict the way targeted coverage states are reached. Initial and final state of the test sequence can be defined and some states or transitions can be forbidden to appear in the test sequence. It is also possible to require some state between another two states in a test suit.

Finally, M μ ALT allows non-deterministic expressions. This 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 supported in M μ ALT.

When the abstract description is ready it is translated to a state machine representation usable by the GOTCHA test generation tool (Figure 1). The intended coverage model is also extracted during this translation from supplementary M μ 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 first search or a breadth first search from each of the start states. Progress reports on this initial state space exploration can be customized in a limited way.

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

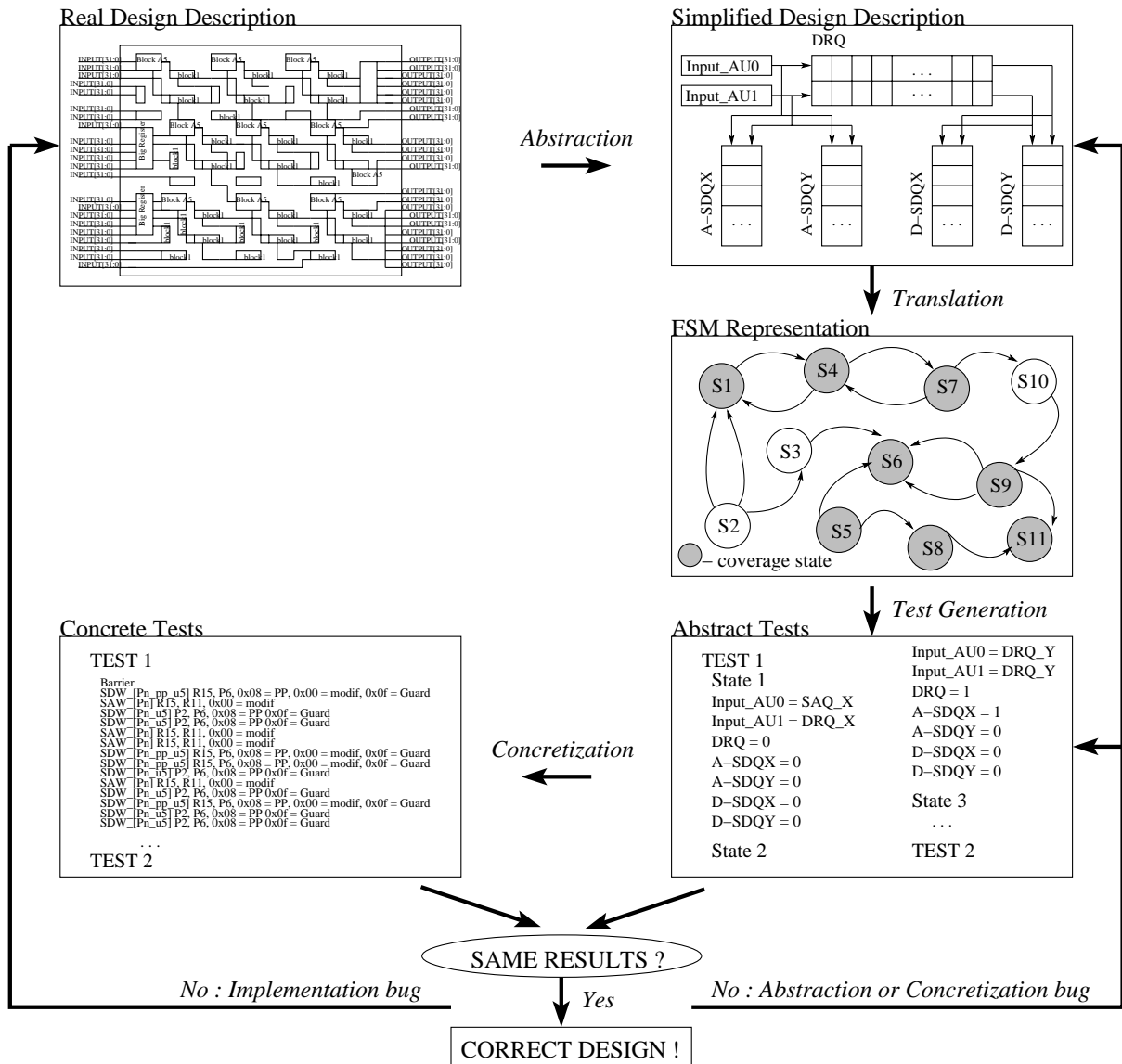


Figure 1. Genevieve test generation methodology

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 1 roughly shows this process. A state is determined by concrete values of all state variables (coverage or not). In the Figure 1 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 1 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 a coverage task. They can not be directly applied to the real design. In order to obtain real or **concretes tests**, we have

to make **concretization** of concrete test inputs. The concretization consists of two major transformations. Firstly, the design variables not corresponding to the design inputs are removed from each state of the abstract test. After this operation an abstract test sequence only contains abstract inputs. Then these input variables are replaced with functions that provide the intended values to real design inputs. Normally, every value of each abstract input variable demands a 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 a functional level. In addition, a preamble and epilogue are almost always required in order to reset the real design before the test and compare the results after the test. Normally the concretization is done by a straightforward translation. At the end of the concretization process real concrete test suites are ready

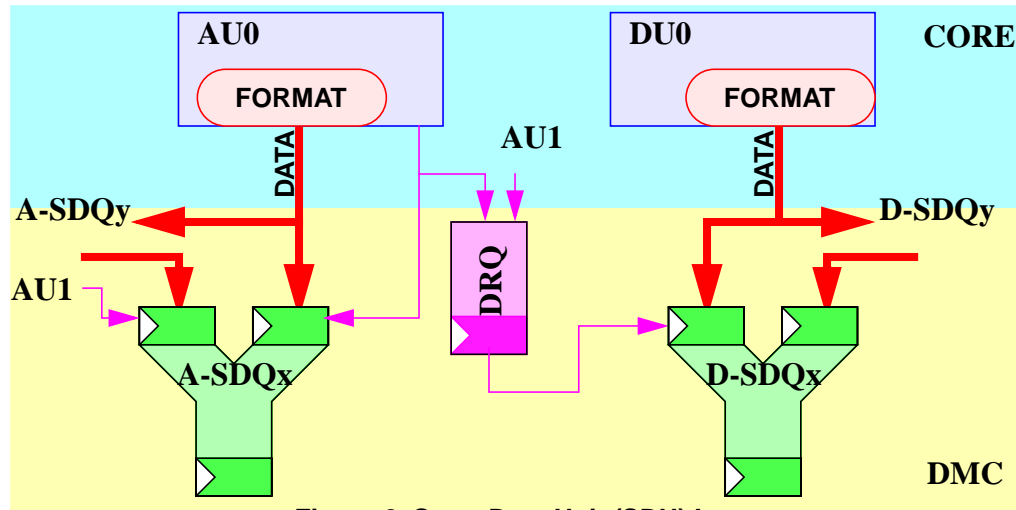


Figure 2. Store Data Unit (SDU) Input

for simulation or emulation.

After simulation/emulation of the real design the real test results have to be compared with expected abstract values. The comparison can be seen as the reverse of concretization: abstract test results are represented by the design state variables. 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 1, 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 the 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 else 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 by cycle matching). 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. The verified SDU block

This section is devoted to a description of the SDU block that was chosen as the first application example of

the Genevieve project. The 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 was 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 2. 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 sub-units (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 can occasionally 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 (bank X or Y):

- 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.

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 2 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. Genevieve test generation

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 a 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 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 behav-

our 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 alone 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 3.

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 data 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 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 de-

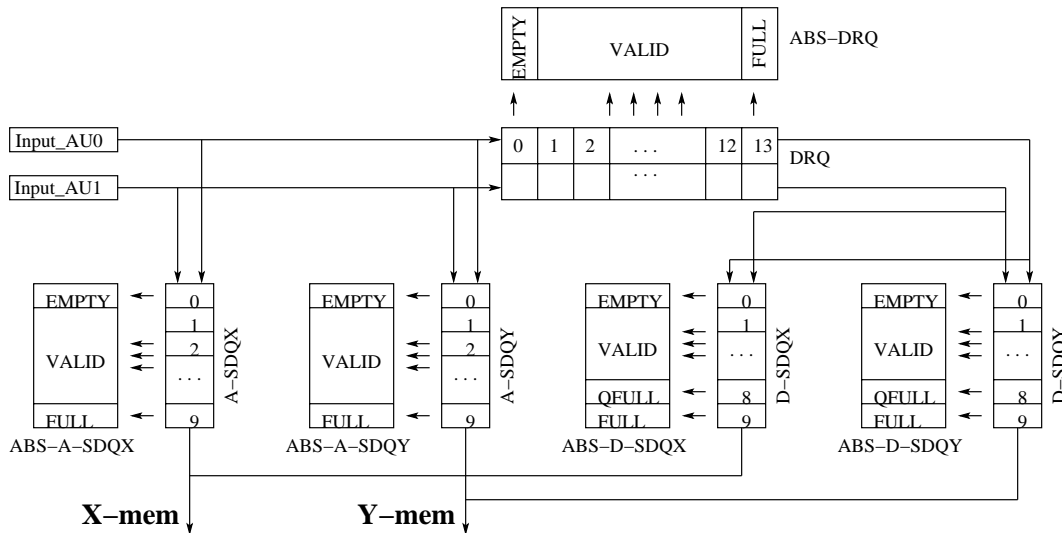


Figure 3. Abstract SDU model

sign, 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-A-SDQX, ABS-A-SDQY, ABS-D-SDQX, and ABS-D-SDQY have the special CVAR_SCALAR attribute), thus giving 432 possible coverage states (see page 4). No refinements of coverage models, like reducing the set of coverage states by means of special functions or defining transitions to cover, were used. No final test states were defined so no final test state is sought once a coverage task is achieved by some test.

The input signals participate in the global state space definition 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. To solve this problem the tool is directed towards interesting corner cases by using guidepost. 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 24 versions of the same abstract model, each differing in in-

put assignments. This then generated test suites that covered some 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 expands the final test suites with new tests.

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, it means 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 on page 2. The concretization process for the SDU unit is shown in Figure 4.

Let’s consider Figure 4 in more detail. The transformation is done in two steps. The first step is translation of an abstract test into a test specification for Genesys (see [6]), a model-based test generator. In every abstract test each assignment of the SDU input variables (AU0 or AU1) is replaced with a Genesys macro supplying desired value to the real SDU 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 basically

- 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 also added at the beginning of each test specification.

The second step is final test generation using Genesys.

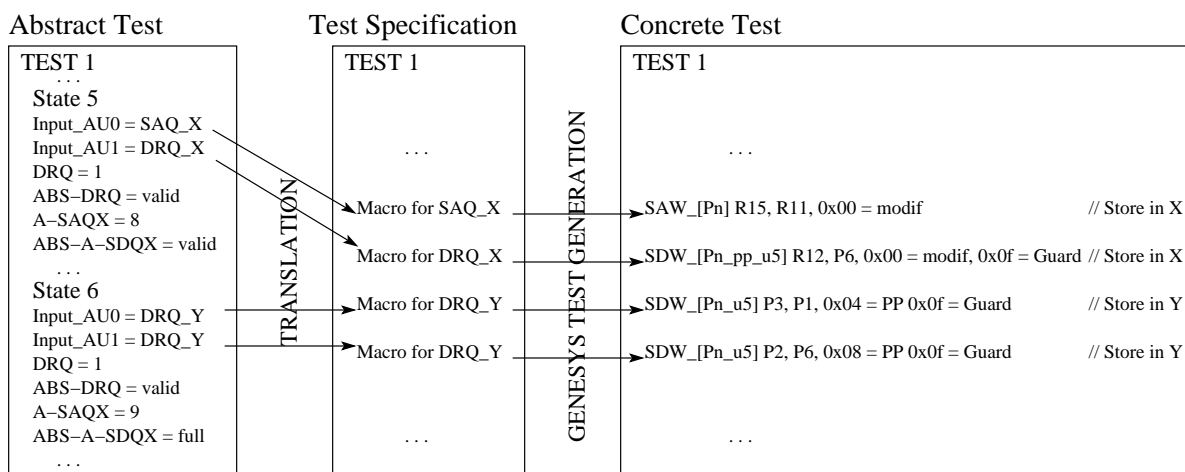


Figure 4. Concretization process for the SDU unit

Each test specification is transformed into an instruction sequence ready for simulation. The macros of instruction sequence are translated by Genesys into concrete ST100 commands. The concrete commands 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.

3.4 Comparison of abstract and concrete tests

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 extracted 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 a separate clock cycle.

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

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

4. Conclusion

This work resulted in an efficient test generation methodology demonstrated on a complex design. We established the different steps of the test generation process and finally obtained concrete tests suitable for real design simulations. We also created associated 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 ($10 \cdot 10 \cdot 10 \cdot 10 \cdot 14 = 140000$ real queue states multiplied by 432 abstract queue states). We could only generate tests by defining much smaller coverage models 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. Before Genevieve the designers and verification team mostly used metrics based on the coverage of the hardware description. In practice it appears that these code-based metrics are very weak and do not cover "border" circuit behaviour.

Based on coverage models we measured the quality of the generated tests. Using the Genevieve methodology we managed to obtain tests for 293 of 432 possible corner cases of interest. During the comparison step we discovered that not all abstract tests matched simulation results: 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. Genevieve and Manual Tests

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

5. References

1. M. Benjamin and all. *A Study in Coverage-Driven Test Generation*, in DAC 99
2. K. L. McMillan *Symbolic Model Checking* Kluwer Academic Press, Norwell, MA, 1993
3. T. Melham *Abstraction Mechanisms for Hardware Verification in VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, January 1987
4. Genevieve: ESPRIT Project 25314 Internal report *Modelling Language Specification*, February 1999
5. D. Geist and all. *Coverage-Directed Test Generation Using Symbolic Techniques*, in FMCAD 96, November 1996
6. A. Aharon and all. *Test program generation for functional verification of PowerPC processors in IBM*, in DAC 95, pages 279-285, 1995