Semi-Formal Test Generation and Resolving a Temporal Abstraction Problem in Practice: Industrial Application

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

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

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

Abstract This document describes a successful application of a semi-formal test generation technique to the verification of Direct Memory Access Controller (DMAC) of ST50, a new general purpose RISC microprocessor developed by STMicroelectronics and Hitachi. Like other memory-related devices, the DMA controller challenges formal techniques because of the state explosion problem. To cope with the challenge, abstraction mechanism is applied during test generation: several abstract models are created in order to verify different functional aspects of the design. We also propose a practical solution to overcome a temporal abstraction problem that arises when tests issued from an abstract model have to be applied during real design simulation.

I. Introduction

This work continues the series of experiments with Genevieve test generation methodology. While the previous work ([1]) deals with a sophisticated environment interface where it is very difficult "to drive" the unit under test to a desired state, this experiment challenges a complex, potentially state explosive internal structure. We describe a methodological solution we found to overcome this difficulty. In particular, we show how multiple abstraction models are applied to generate tests and how we resolved a temporal abstraction problem that arises when tests issued from an abstract model have to be applied during real design simulation.

II. Genevieve Test Generation Methodology

The Genevieve methodology ([2]) relies on formal methods ([3]) to generate test suites for specific behaviour of the design under test. A specific behaviour, often called as *corner case*, is a composition of border behaviours for different design parts or blocks. In this documents we use "corner cases" to specify particular design states we want to test.

To cope with state explosion, we describe the design under test in a simplified manner. This process, called *abstraction*, is shown in Figure 1. While there exist different kinds of abstract mechanism ([4]), 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* to group irrelevant data into classes or ignore them;
3. *temporal abstraction* to consider the events order, rather than precise timing.

We use the Mμ ALT (Modelling micro-Architecture Language for Traversal) language for abstract description of the design under test (see [5]). 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 determined by adding special attributes to "interesting" signals or variables which are called

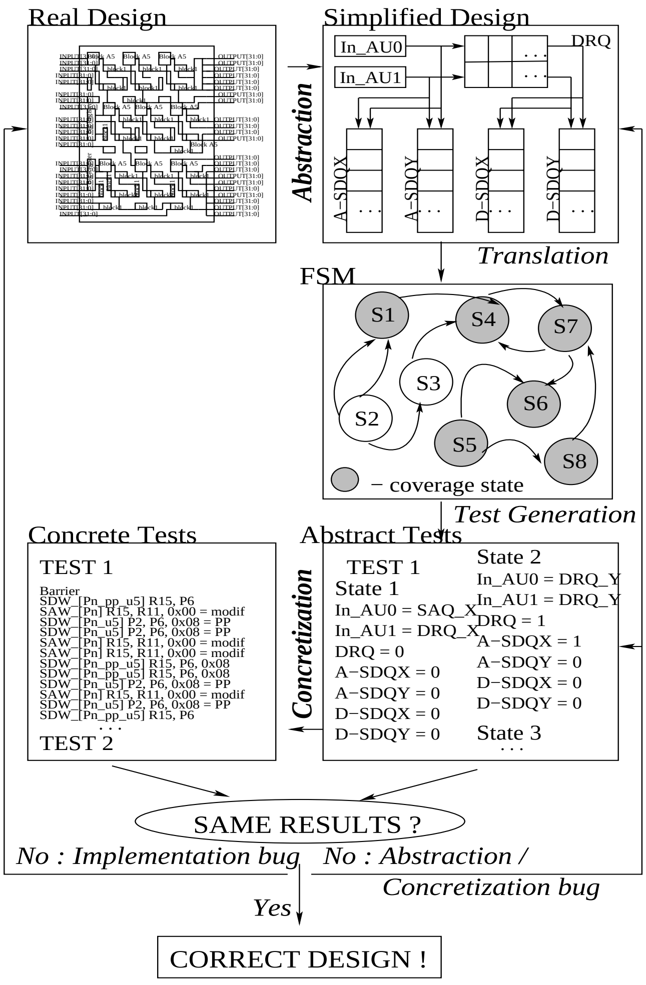


Fig. 1. Genevieve test generation methodology

coverage variables. 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 an FSM model. Later in this document we use the term "state" to refer to a combination of variable values and we say that coverage model consists of coverage states. The coverage model can be further refined by means of special functions. Thus, the designer might be interested to test the circuit only with some specific values of coverage 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 force a state to be obligatory between other two states in a test suite.

Finally, the 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 provided by the Mμ ALT special constructions.

When the abstract description is ready, it is translated to a state machine representation used by the GOTCHA test generation tool (Figure 1). 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 specifically for the Genevieve project ([6] , [7]).

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 algorithm from each of the start states.

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) then continuing on to a final state. If the test length recommendation has been exceeded at this point, then the test is output, else 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, GOTCHA results in a set of *abstract tests*, each abstract test containing a sequence of states. Figure 1 shows this process. A state is determined by concrete values of all state variables (coverage or not). The design state variables contain both state variables in a proper sense (it means variables or signals that represent design registers) as well as input state variables. This is because the behaviour of the environment, and therefore the inputs of FSM, depends on reaction of the design under test. It is then necessary to model the environment as one or more FSMs that provide legal input to the design.

The environment behaviour is integrated into the design model, thus extending the state set of the corresponding FSM and eliminating its input set. This situation is similar to that during testing with testbenches. The outmost testbench circuit contains no inputs. If every variable, signal or register is considered as a state variable, the FSM corresponding to the outmost testbench contains no input but only state set.

Abstract tests give sequence of states to reach a coverage task. However, 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 in translating of abstract tests into the tests suitable for a simulation or emulation environment. In general, the translated tests have to provide intended abstract values to real design inputs. For this reason the concretization can be considered as related to the input state variables: only these variables are taken into account during translation process.

The level and structure of concrete tests depend on test objectives. It may be simulator commands supplying values to the design inputs or microcontroller instructions if the design is tested at functional level. In addition, a prologue and epilogue test suites are required in order to reset the real design before test and correctly finish the test.

After simulation of the real design, obtained real test results have to be compared with expected abstract ones. The *comparison* can be seen as opposite to the concretization: it is related to the state variables in a proper sense. Only these abstract variables are compared against concrete test results. As abstract and real design descriptions can differ considerably, the relation between abstract and real state variables must be established.

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 signal/variable 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 is not used in abstract design description, then successive states of abstract test correspond to successive states of the real design. Otherwise, supplementary states 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 abstract model is wrong or too diverted from the original device.Third, the concretization does not supply intended abstract inputs to the real design. In each case necessary modifications must be made and the whole process has to be repeated.

III. Description of the Verified DMA Controller

The ST50 belongs to a series of products conjointly developed by STMicroelectronics and Hitachi. This is a new architecture that enables a family of low-cost, small, high-performance microprocessors. It has simple and efficient hardware implementation specifically designed to be an excellent target for compilers.

The ST50 integrates an on-chip direct memory access controller (DMAC) that has been chosen as an application example of the Genevieve test generation methodology. The DMAC can be used in place of the CPU to perform high-speed data transfers between memory and/or memory-mapped internal devices.

The DMAC architecture is shown in Figure 2. It consists of two common registers and four channels, each channel being able to perform independent memory transfer.

The memory transfer associated with a channel is defined by the channel configuration which is composed of the following elements:

- *counter register* defines the number of transactions within one memory transfer;

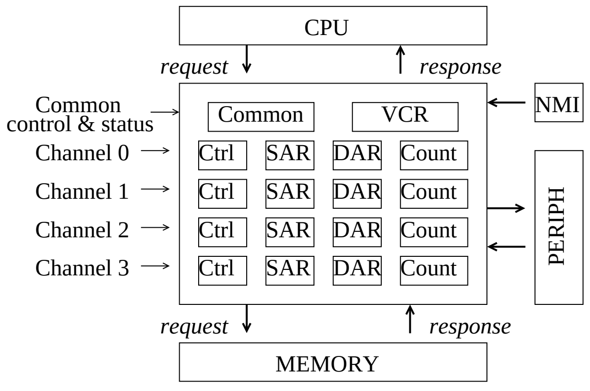


Fig. 2. The DMAC architecture

- *source address register* and *source address incrementing mode* are used as the address from which the next transaction unit will be fetched; the source address will be incremented or decremented as DMA channel proceeds in accordance with the source address incrementing mode;
- *destination address* register and *destination address incrementing mode* are used as the address at which the next transaction unit will be stored; the destination address will be incremented or decremented as DMA channel proceeds in accordance with the destination address incrementing mode;
- *transfer size* defines the size of one transaction that can be 1, 2, 4, 8, 16 or 32 bytes; if the transfer size is 16 bytes and the counter register is set to 10, then 16*10 bytes will be transferred within the memory transfer associated with the channel;
- *resource selection* defines the memory transfer source and can be either in *auto request mode* or in *on-chip peripheral request mode*; in the auto request mode the source of memory transfer is the main memory and transfer request signal for each memory transaction is generated by the DMAC automatically; in on-chip peripheral request mode the source of memory transfer is one of six peripheral devices and transfer request signal for each memory transaction is generated by the DMAC only if the associated peripheral device is ready to provide data.

In addition, each channel can be individually suspended by resetting corresponding *transfer enable [0 ... 3]* bit. This bit is also used to start a channel transfer when the channel programming is finished.

The common registers are used for general DMAC configuration (the register "common") and for stroring the DMAC status after transfer completion (the register "vcr"). The *master enable* bit of the register "common" allows to simultaneously enable or disable all four channels. The DMAC total suspension also occurs if the NMI (Not Maskable Interrupt) signal is set. The channel priority ordering is defined by the *priority* bit of the register "common". The DMAC can transfer data in two priority modes:

- *fixed priority mode*: the relative channels priority remains fixed: channel 0 -> channel 1 -> channel 2 -> channel 3, channel 0 being the highest priority channel; all channels operate in a steal mode meaning that the lower priority channel can steal control form the higher priority channel if the channel is idle; the higher priority channel regains or loses control depending on the speed at which the serviced unit requests the DMAC. Thus, if channel 0 is programmed to service in auto request mode then channel 1 gets control only when the transfer on channel 0 is completed;
- *round robin mode*: in this mode each time the transfer of one transfer unit (1, 2, 4, 8, 16 or 32 bytes) ends on a given channel, that channel is assigned the lowest priority level; if a channel is programmed in on-chip peripheral mode and the associated peripheral is not ready for transaction, then the priority moves to the next channel.

IV. How Abstraction is Applied to Generate Tests

Because the DMAC is a complex device and it is impossible to check all possible transfer combinations, its abstract model has to be carefully chosen in order to hide irrelevant details and at the same time to catch the main DMAC properties. As the Genevieve methodology employs a "model checking" based tool, it is essential to eliminate a potential source of state explosion in the abstract model. From this point of view

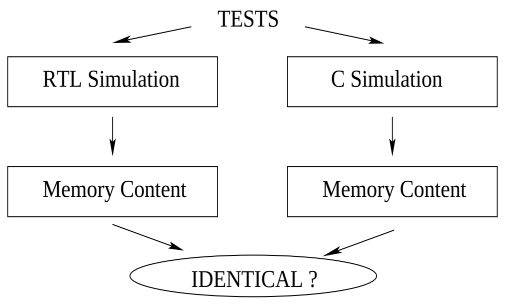


Fig. 3. Memory comparison

the most challenging DMAC parts are, of course, the address and data buses.

That is why we chose not to check the memory transfer itself (i.e. whether the intended data is written at the intended address) by the tests generated with Genevieve. To ensure, however, that the main purpose of the memory transfer is achieved, we compare the memory contents obtained after RTL and C simulation with the Genevieve tests. If the memories are identical, the main transfer mechanism of the DMAC is considered correct. This process is schematically shown in Figure 3.

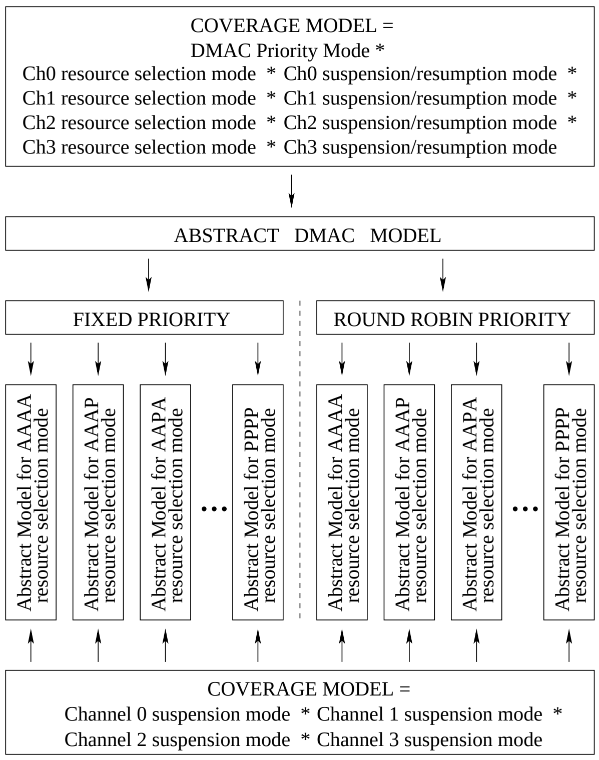
After the elimination of "state explosive" address and data, the DMAC behaviour is still very complex: the possibility to separately program each channel results in a huge number of different DMAC configuration, each of them leading to different DMAC behaviour. In the abstract model we decided to concentrate on the correct order of memory transactions according to the priority mode, channels resource selection, and the presence of peripheral requests if channels are programmed in on-chip peripheral mode. The equally important issue for the DMAC verification is whether suspension and resumption of one or several channels affects correct memory transfers.

Thus, the coverage model in the abstract DMAC description was defined as all possible combinations of the DMAC priority mode, each channel resource selection mode and each channel suspension/resumption mode defined by the *transfer enable[0 ..3]* signal. Other DMAC registers such as channels transfer size were defined randomly during the translation step from abstract tests to concrete ones.

However, even if the coverage model is restricted to most important cases, the corresponding abstract model is still very big for GOTCHA if we try to cover all coverage tasks at once. A solution consists in splitting the overall coverage model into several sub-models. In other words, we wrote different abstract DMAC models, each abstract model describing a particular aspect of the DMAC functionality and resulting in abstract tests for that particular functionality. This methodological decision is illustrated by Figure 4.

The initial coverage model is schematically shown as a multiplication of interesting DMAC modes, thus giving 2^9 = 512 different coverage tasks (we consider that each channel can be either in auto request or on-chip peripheral mode; a concrete peripheral associated with the channel will be randomly chosen during the translation step of abstract tests into real ones).

To generate tests for the defined coverage tasks, two big abstract submodels were created corresponding to two well-distinguished different behaviours of the DMAC for each of its priority modes. Each submodel was then further refined into abstract models that correspond to the DMAC behaviour with particular combination of resource selection modes. For example, the abstract model for the AAPA resource selection mode (Figure 4) correspond to the DMAC behaviour when the



COVERAGE MODEL =
DMAC Priority Mode *
Ch0 resource selection mode * Ch0 suspension/resumption mode *
Ch1 resource selection mode * Ch1 suspension/resumption mode *
Ch2 resource selection mode * Ch2 suspension/resumption mode *
Ch3 resource selection mode * Ch3 suspension/resumption mode
ABSTRACT DMAC MODEL
FIXED PRIORITY
ROUND ROBIN PRIORITY
Abstract Model for AAAA resource selection mode
Abstract Model for AAAP resource selection mode
Abstract Model for AAPA resource selection mode
Abstract Model for PPPP resource selection mode
COVERAGE MODEL =
Channel 0 suspension mode * Channel 1 suspension mode *
Channel 2 suspension mode * Channel 3 suspension mode

Fig. 4. Splitting DMAC into abstract models

channels 0, 2 and 3 are programmed to transfer in auto request mode, and the channel 1 is programmed to transfer in on-chip peripheral mode (with any available peripheral device). While the two submodels corresponding to the priority modes distinguish naturally, the difference between abstract models corresponding to various combinations of resource selection modes is minimum and basically consists in commenting fragments of the common code. So, it did not cost considerable efforts to create 16 abstract models for each priority mode.

The final 32 abstract models are reasonably small. Moreover, the coverage model for every final abstract model is also small and contains only 16 combinations of suspension/ resumption state of four channels. Thus, abstract tests were easily created by GOTCHA for the final abstract models.

Another challenge we had to confront while generating abstract tests is the complex pipeline mechanism that implements DMAC memory transfers. Each memory transaction is divided into two parts: firstly, a data has to be read from the source address and, secondly, it has to be written at the destination address. As the DMAC uses the same bus for read and write procedures, a well-established order does exist to control access to the bus. Very roughly, a two-depth pipeline corresponds to each read and write procedure. The read pipeline is filled first. Then, the write pipeline is filled in the same order of transactions as the read pipeline. When a write procedure ends one memory transaction, a place is liberated in the read pipeline. A new read request of the next memory transaction can enter the read pipeline. A simplified version of this mechanism is shown in Figure 5. The transactions 1, 2, 3, etc. can belong to any DMAC channel.

In addition to the complex pipeline implementation, the actual impact of some signals such as peripheral transfer request is not immediate but delayed by several clock cycles. For example, in Figure 5, the read procedure of the transaction 3 will belong to the channel 2 (assuming channel 2 has the highest priority at the beginning of the read procedure). However the read procedure of the transaction 4 will not belong to

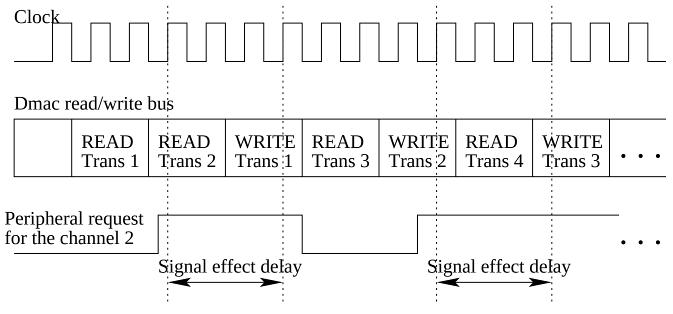


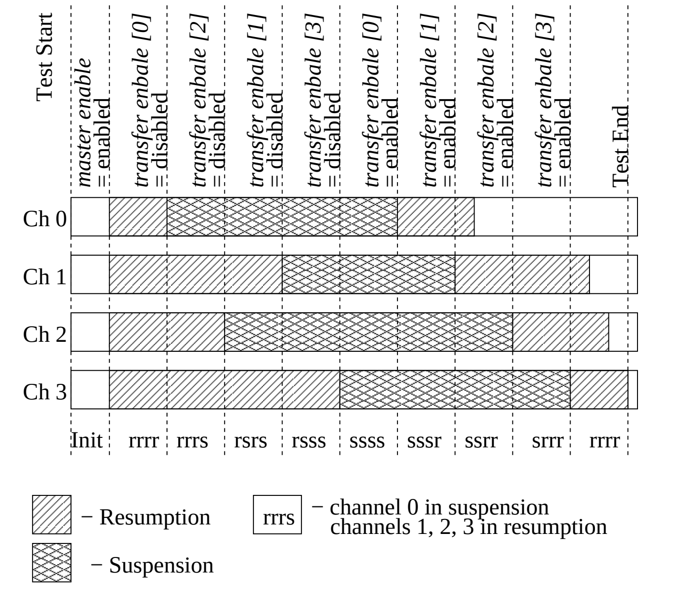
Fig. 5. Pipeline mechanism

the channel 2 due to a small delay between the peripheral request signal and the transaction 4.

To cope with this problem, we have chosen a fixed test scenario in which the order of read procedures is predictable. This order of read procedures is the subject of the verification: each read procedure in a sequence of memory transactions must belong to a specific channel according to the current DMAC configuration (priority mode, resource selection, etc.).

According to the scenario, the tests start with the programming of counter registers. Then all channels are simultaneously resumed by enabling the common *master enable* signal. After the channels complete more than half of intended memory transactions, they are randomly suspended by disabling corresponding *transfer enable [0..3]* signal, thus trying to cover the coverage tasks. The tests end by resuming channels one by one until all memory transactions on all four channels are completed. An example of abstract test is shown in Figure 6. This particular test covers 8 coverage tasks, i.e. 8 combinations of suspension/resumption channels mode.

The test scenario (including initialisation stage, random signal assignments and final state definition) is integrated in the final abstract models of Figure 4. These models resulted in generation of 273 abstract tests, 126 tests for the fixed priority mode and 147 for the round robin priority mode. The intended coverage tasks were covered for each particular abstract model. The next section will discuss the problem encountered while translating abstract tests into concrete ones, suitable for real simulation.



Test Start
master enable = enabled
transfer enbale [0] = disabled
transfer enbale [2] = disabled
transfer enbale [1] = disabled
transfer enbale [3] = disabled
transfer enbale [0] = enabled
transfer enbale [1] = enabled
transfer enbale [2] = enabled
transfer enbale [3] = enabled
Test End
Ch 0
Ch 1
Ch 2
Ch 3
Init
rrrr
rrrs
rrsrs
rsss
ssss
sssr
ssrr
srrr
rrrr
– Resumption
– Suspension
rrrs
– channel 0 in suspension channels 1, 2, 3 in resumption

Fig. 6. Example of abstract test

V. Translation of Abstract Test: Solving the Temporal Abstraction Problem

The abstract DMAC models make the full use of abstraction mechanisms: functional as well as temporal. While the functional abstraction is not difficult to deal with during translation process of abstract tests (a simple matching function is required), the temporal abstraction changes the time scale of the abstract model with respect to original design. The supplying of inputs in concrete tests becomes problematic.

Indeed, abstract tests contain abstract variables that correspond to the design inputs and that have to provide intended design inputs during actual simulation. As the abstract model is not cycle accurate, it is not clear when the abstract inputs must be supplied during concrete tests because an unpredictable number of additional real states can be inserted between initial abstract states. Such a situation is shown in Figure 7: we consider that a state corresponds to a clock cycle.

To resolve the problem, a simulation system is needed that dynamically monitors the internal DMAC behaviour and once a concrete state that corresponds to the desired abstract state is reached, generates required concrete inputs. However, to make the exact matching between abstract and concrete states is a very difficult, if not impossible, task. Moreover, known simulation tools require static simulation input file. How to avoid all these difficulties?

Bergamachi and Raje ([8], [9]) address a similar problem that appears when RTL-level implementation obtained after behavioural synthesis has to be compared against algorithmic specification. The problem is solved by using so-called Observable Time Windows - the instants where the comparison between high-level specification and RTL implementation is appropriate. The Observable Time Windows are identified during simulation with the help of additional hardware.

We propose a different, software based approach. The translation of abstract tests is realized as follows: each abstract input supply is replaced with a fragment of simulation language code that provides intended input values during actual simulation. Each abstract output observation is replaced with a fragment of simulation language code that compares actual concrete outputs with the expected ones. In addition, the output code fragment is waiting (in terms of used simulation language) for the expected event to happen and is blocking further simulation process. For example, the inputs of the abstract state 4 will not be supplied during the concrete test until the expected outputs of the abstract state 3 are observed during the simulation (Figure 7).

The proposed concept consistently continues the Genevieve idea of integrating the environment stimulus (device inputs) into the device model: the inputs are assigned depending on the reaction of the device during test generation, simulation and comparison process. It has to be noticed, that the concretization and comparison steps are merged together thus simplifying the whole verification process: if all expected results of an abstract test match with the results obtained during simulation, then the actual simulation corresponding to the abstract test terminates normally, else abnormally.

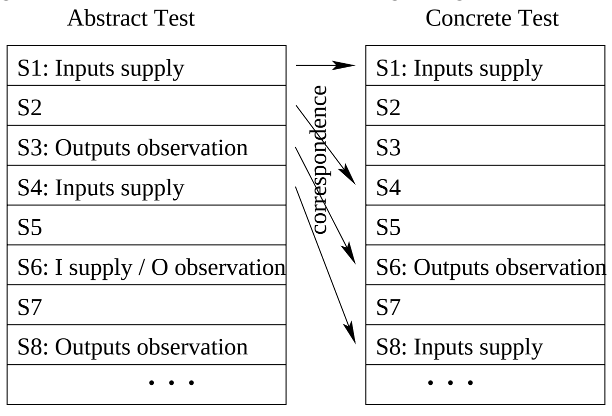


Fig. 7. Temporal abstraction problem

The chosen translation approach also allows the simulation input file to be created *before* the simulation starts. Even if the device inputs are supplied "on the fly", the procedures supplying the inputs are written in the terms of simulation language and translated in advance from abstract tests. It makes the simulation possible because current simulation tools require complete input simulation file before the simulation itself.

A QuickBench simulation environment provides all the facilities necessary to implement the translation and comparison step of the verification process. QuickBench is a conventional simulator (vcs) with additional capacities to model the device environment by the means of the *rave* language specific to QuickBench. This feature is achieved by compiling together the rave interpreter with the Synopsys Verilog.

The crucial property we need for Genevieve is implemented by so-called *rendez-vous* construction. A rendez-vous is similar to a synchronization point of two processes: a process suspends when achieving the synchronization point and can only resume when the counterpart process achieves the same synchronization point. Similarly, in a test suite we use the rendez-vous commands in order to wait for expected results: further simulation process is frozen until the expected results are received. During the simulation a special rave procedure is monitoring the DMAC outputs and when an expected event happens, it provides the counterpart rendez-vous construction necessary to unfreeze the simulation process. As we mentioned earlier, the expected events are *read* requests that DMAC sends to memory or peripherals as the beginnings of memory transactions.

An example of pseudo-concrete simulation file translated from the abstract test of Figure 7 is shown in Figure 8. The test prologue and epilogue are required to reset the design before the test and successfully finish the simulation after the test. If an abstract state contains both input supplies and output observation, then firstly, the expected outputs are observed and, secondly, the intended inputs are supplied (as for the abstract state 6 in Figure 8). This is the logical events order because inputs cause the design to move from current state to its next state and we need to check whether the design achieved the current state before new inputs are supplied.

If the simulation of QuickBench test is successful (i.e. it ends with the message "Test finished successfully" in Figure 8), this indicates that the design satisfies the functionality verified by the corresponding abstract test. All 273 abstract tests generated for the DMA controller were successfully translated and simulated, thus proving the correct RTL implementation of the DMAC device. However, we found a bug in its C-model: while ultimate memory transfers were correct, the order of memory transactions was not always consistent with the order of memory transactions in RTL implementation.

VI. Conclusion

This work resulted in successful verification of the Direct Memory Access Controller traditionally considered very difficult to check by both conventional simulation and formal methods due to the state explosion problem. The application of the Genevieve methodology allowed us to avoid usual difficulties and generate tests for this complex device.

By taking a real-life industrial design we showed the feasibility of test generation using formal methods. We also demonstrated how abstraction mechanism can be applied in

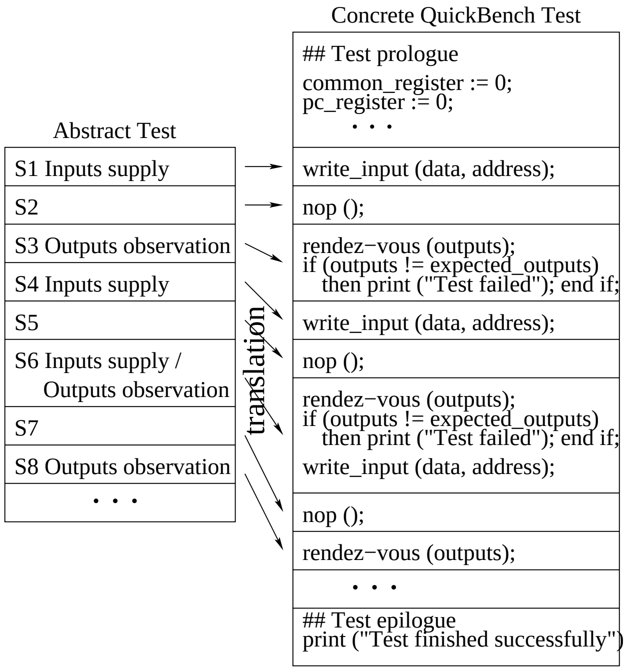

Concrete QuickBench Test
## Test prologue
common_register := 0;
pc_register := 0;
Abstract Test
S1 Inputs supply
S2
S3 Outputs observation
S4 Inputs supply
S5
S6 Inputs supply / Outputs observation
S7
S8 Outputs observation
translation
write_input (data, address);
nop ();
rendez−vous (outputs);
if (outputs != expected_outputs)
then print ("Test failed"); end if;
write_input (data, address);
nop ();
rendez−vous (outputs);
if (outputs != expected_outputs)
then print ("Test failed"); end if;
write_input (data, address);
nop ();
rendez−vous (outputs);
## Test epilogue
print ("Test finished successfully")

Fig. 8. Concretization of abstract tests

practice to generate tests: we created several abstract models (instead of only one), each of them specifying a particular sub-functionality of the device and resulting in the test generation for that particular sub-functionality. Such an approach reduces the complexity of the test generation process while still dealing with various aspects of the design behaviour.

An abstract model, while still complex for test generation, can hide too many details and result in abstract tests not matching concrete design. A methodological solution we propose consists in a specific test *scenario* implying the same event order for both abstract model and real design. The test scenario is integrated in the abstract model thus guiding the test generation towards abstract tests consistent with the real implementation.

Another important result of this work is the solution proposed for the temporal abstraction problem. The temporal abstraction changes the time scale of the real design and loses the exact time of input supply during abstract test generation. If, however, the input supply is alternated with the expected output observation, then the event order intended in abstract tests is maintained. Among existing simulation tools, the QuickBench simulation environment allows to implement such an alternation. It has to be noticed though, that this simulation tool is quite complex and more user-friendly simulators providing the same features are required.

During this work we created 32 abstract models of the DMA controller that generated 273 abstract tests. The abstract tests covered all 512 coverage tasks defined by the coverage model. The abstract tests were translated into concrete QuickBench tests and successfully simulated. A bug in the DMAC C-model was found as a result of the verification process.

To estimate the quality of the generated tests we compared them with the tests created manually for the same device. The results are given in the table below.

TABLE I COMPARISON: GENEVIEVE COVERAGE MODEL

	Genevieve	Manual
Test number	273	156 (from 3000)
Covered corner cases	512	41

From several thousands manual tests (3000), only 156 were devoted to the verification of the “dynamic” behaviour of the DMAC, i.e. when channels are alternatively switched on and off during data transfer. All the other tests were written to check the correct DMAC programming (notably possible configurations of different transfer size associated with each particular channel) and would not improve the overall coverage.

We can see that more significant numbers of manual tests covered only 41 corner cases. This is because the manual tests did not even attempt to check the behaviour we were interested in during the verification with Genevieve. It is inconceivable to create the same quality manual tests as we created with Genevieve because of the huge number of functional modes of the DMAC device. Genevieve, on the contrary, automatically generates tests for the coverage model defined by simple enumeration of coverage variables.

Even if the measurement results given above are more than satisfactory, we wanted nevertheless to compare Genevieve and Manual tests using some “neutral” metrics. In fact, it is very difficult to determine the advantage of manual tests with respect to Genevieve ones: manual tests could cover corner cases that Genevieve tests did not cover. We chose VHDL coverage model as this neutral metrics: the number of VHDL statements or VHDL conditional branches covered by tests during simulation is used for the test comparison. The VHDL coverage was done using Verification Navigator from TransEda.

TABLE II COMPARISON: VHDL COVERAGE MODEL

	Genevieve	Manual
Total number of tests	273	3000
Covered VHDL Statements (%)	86.8	92.1
Covered VHDL Branches (%)	75.0	85.5

The results are reported in Table II: given the number of Genevieve tests that order of magnitude less than the number of manual tests, the Genevieve tests still show the VHDL coverage competitive with manual tests. This fact is very encouraging and indicates the quality of Genevieve tests.

VII. REFERENCES

[1]. J.Dushina, M.Benjamin and D. Geist *Semi-Formal Test Generation with Genevieve*, in DAC 01

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

[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]. Genevieve: ESPIRIT Project 25314. *Modelling Language Specification*, February 1999.

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

[7]. A. Aharon and all. *Test program generation for functional verification of PowerPC processors in IBM* in DAC 95, pages 279–285, 1995.

[8]. Reinaldo A. Bergamaschi and Salil Raje *Observable Time Windows: Verifying the Results of High-Level Synthesis* in ED&TC’96, pages 350-356, 1996

[9]. Reinaldo A. Bergamaschi and Salil Raje *Observable Time Windows: Verifying High-Level Synthesis Results* in IEEE Design & Test on Computers, pages 40-50, April-June of 1997