Hardware testing using a communication protocol conformance testing tool

Hakim Kahlouche[2], César Viho[1], and Massimo Zendri[3]

[1] Irisa/IFSIC-Université de Rennes I, Campus de Beaulieu,
F-35042 Rennes Cedex, France
E-mail: `viho@irisa.fr`, WWW home page: `http://www.irisa.fr/pampa`
[2] Inria, Campus de Beaulieu, F-35042 Rennes Cedex, France
[3] Bull R&D, Via ai Laboratori Olivetti, I-20010 Pregnana, Milanese (MI), Italy

Abstract. We present an experiment which has demonstrated that methods and tools developed in the context of black box conformance testing of communication protocols can be efficiently used for testing the cache coherency protocol of a hardware multi-processor architecture. We have used the automatic conformance tests generator TGV developed by INRIA to generate abstract tests and we have developed a software in order to make them executable in the real test environment of Bull.
The TGV approach has been considered by the hardware testing community as a serious alternative to usual random test generation. It overwhelms the well known debugging and coverage problems linked to this kind of technic.

1 Introduction

In this paper, we are concerned with the so called *conformance testing* which consists in testing whether an implementation of a system behaves as described in its specification. According to the domain, there exists different kinds of methods and tools dedicated to conformance testing. In some cases, one can find some similitude between these different methods. This is the case for hardware off-line testing and communication protocol conformance testing in the experiment we are describing in this paper.

On one side of this end-to-end experiment done in the context of the VASY (Validation of Systems) action within the Dyade / Bull-Inria R&D Joint Venture, there were engineers of Bull using their usual methodology to develop a multi-processor architecture called in the following the Bull's CC_NUMA machine. In hardware design the description of the system is often based on hardware description languages such as VHDL [1] or VERILOG [2]. This is due to the ability of these languages to describe various levels including hardware-related details such as register-transfer, gate and switch levels. In the case that one is particularly interested with high-level functionalities, such as Cache Coherency Protocols, these details may lead to over-specification. Even though, there are different abstraction levels (VHDL behavioral style), abstract synchronization mechanisms such as rendez-vous are not easily available. Moreover, the tools applied on the specifications (for verification, test generation,...) may be unusable since they are needlessly complex. Therefore, one may wonder whether the formal specification languages and associated tools designed in another analogous domain (like computer network area) could be better suitable for the description and test of high-level functionalities [3]. We have chosen the LOTOS language for the formal specification of the BULL'S CC_NUMA architecture because its underlying semantics model is based on the rendez-vous synchronization mechanism which is well suited for the specification of hardware entities [4] such as processors, memory controllers, bus arbiters, etc. The communications between these components by sending electrical signals on conductors are easily described by interactions between LOTOS processes. Another reason of this choice is that LOTOS is a standard language well known in computer network community and often used for the description of communication protocols.

On another side, the prototype TGV has been developed by Inria-Rennes to generate test cases for communication protocols using the black box conformance testing approach. The main purpose of TGV is to fit as well as possible the industrial practice of test generation. Given a formal specification of the system to be tested and a formal description of a test purpose (which represents an abstract form of the property to be tested), TGV generates an abstract test case. It is a direct acyclic graph in which each path represents a test sequence with associated verdict which indicates whether the implementation under test (IUT) conforms with the specification or not [5]. A test case generated by TGV is *interactive* because each sequence is series of interactions between the tester and the IUT. and an output of the tester depends on what it has previously observed from the IUT. Notice that this is not the case in hardware testing which is rather "batch": after stimulating the IUT, one observes its reactions and analyzes them afterwards. TGV has been experimented on the Drex military protocol [6] and on the SSCOP protocol [7]. The comparison of the hand written test cases with those automatically generated by TGV, has shown its interest and efficiency.

The deal in the experiment described in this paper consists in demonstrating that the TGV tool which has been developed for conformance testing of communication protocols can also be efficiently used to generate tests for hardware architectures. In a first step of this experiment, we have proved that the testing activity done by hand can be automatically done using TGV approach [9]. The main contribution of the results presented in this paper lies in the fact that in this second and final step of our experiment, we have also demonstrated that: *"the interactive nature of conformance testing with* TGV *is advantageous for hardware testing, because it improves the quality of the tests and the test coverage"*.

The following section describes the Bull's CC_NUMA machine, its architecture, its cache coherency protocol, the test purposes and the hardware testing methodology habitually used. In the third section, we present the approach used to make possible the automatic generation of tests with TGV: formal specification and verification of the CC_NUMA cache coherency protocol, formalization of the test purposes. The tools developed in order to make executable the generated tests in the real test environment of Bull are presented in section 4. This is followed in section 5 by the description of the advantages brought by this experiment to both of the two communities (network protocol conformance testing and hardware testing) and a quantitative and qualitative analysis of the results. The conclusion gives some ideas on current and future work.

2 The Bull's CC_NUMA machine: architecture and testing environment

2.1 The general architecture and the cache coherency protocol

The BULL'S CC_NUMA architecture is a multiprocessor system based on a Cache-Coherent Non Uniform Memory Architecture (CC-NUMA). It is derived from Stanford's DASH multiprocessor machine and consists of a scalable interconnection of up to 8 modules (see figure 1).

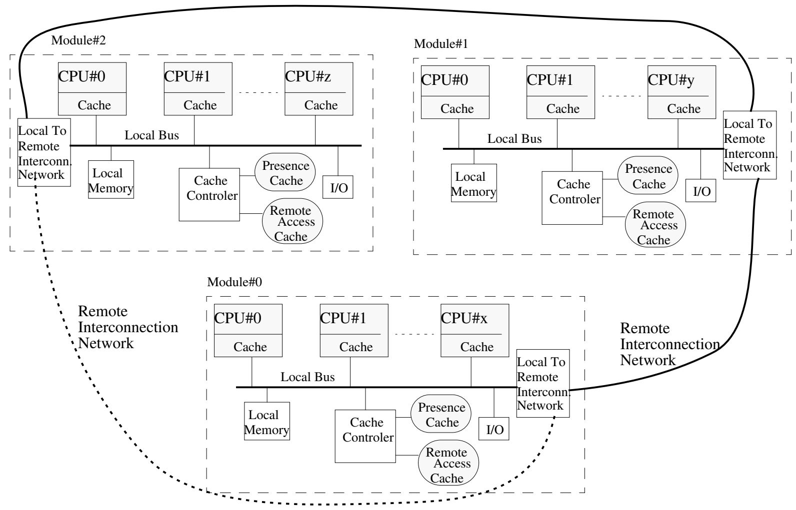

Fig. 1. The BULL'S CC_NUMA General Architecture

The memory is distributed among different modules. Each module contains a set of up to 4 processors. The key feature of the BULL'S CC_NUMA architecture is its *distributed directory based cache coherency protocol* using a Presence Cache and a Remote Cache in each module. The Presence Cache of a module is a cached directory that maps all the blocks cached outside the module. The global performance of the BULL'S CC_NUMA architecture is improved through the Remote Cache (RC) that locally stores the most recently used blocks retrieved from remote memories. Remote memory block can be in one of the following status: uncached, shared, modified which correspond to the possible RC status: (INV)alid, (SH)ared, (MOD)ified.

Thus, testing the Cache Coherency Protocol comes to *verifying that the status of the Presence Cache and Remote Cache are always correctly updated during the execution of any transaction in the* BULL'S CC_NUMA *architecture.*

2.2 The test purposes

The experts of Bull including the designers of the BULL'S CC_NUMA architecture who know its weak points have written a document called the "test plan document". It contains informal description (in the shape of tables with comments) of the main test purposes to be applied to the BULL'S CC_NUMA architecture. Seven *Test Groups* have been identified. In our experiment, we were interested in two Test Groups (Group 3 and 4) concerning the test of the Cache Coherency protocol. An example of test purpose describing an address collision situation is: *"The Module#1 requests for a FLUSH transaction on the block address A0. The block address A0 is in Module#0. Verify that the Module#0 accepts the incoming FLUSH transaction. The CPU#0 of Module#0 executes a RWITM on the same address. Check the immediate address collision on block A0. Check also that the correct response is given by Module#0 and verify the good completion of the FLUSH transaction."*

2.3 The current testing architecture

The testing environment of BULL'S CC_NUMA architecture used in this experiment is called SIM1 environment and is described in Figure 2. It consists of 3 modules, connected on a Remote Interconnection Network. Each module is composed by Processor Behavioral Models (MPB Bus Model), Memory Array and Memory Controller, Arbiter and I/O Block, Coherency Controller, Remote Cache Tag that contains the Tag of Remote Cache and the Presence Cache. The simulation environment is composed of the description of the system (the 3 modules), the kernel event simulator (VSS kernel: VHDL Synopsys Simulator) and a front end human interface (VHDL Debugger). The MPBgen application converts the MPB input commands format (*input files*) into the expected intermediate format (*input tables*) readable by the MPBs. The probe VHDL module is then in charge of down-loading the desired (among the observed) output events; the VSS writes them in a file (PROBE.OUT file).

From testing point of view, the system under test (called SUT on the figure 2) is seen as a black box. Thus, testing in this environment consists in specifying the input files and analyzing the probe output files.

The input files: There is one input file per MPB and an input file describes a sequence of transactions to be executed by one CPU. The input files are currently written by hand according to the informal test purposes specified in the test plan document. The main difficulty in describing these files is the synchronization of the CPUs. The synchronization of all the transactions which are to be executed

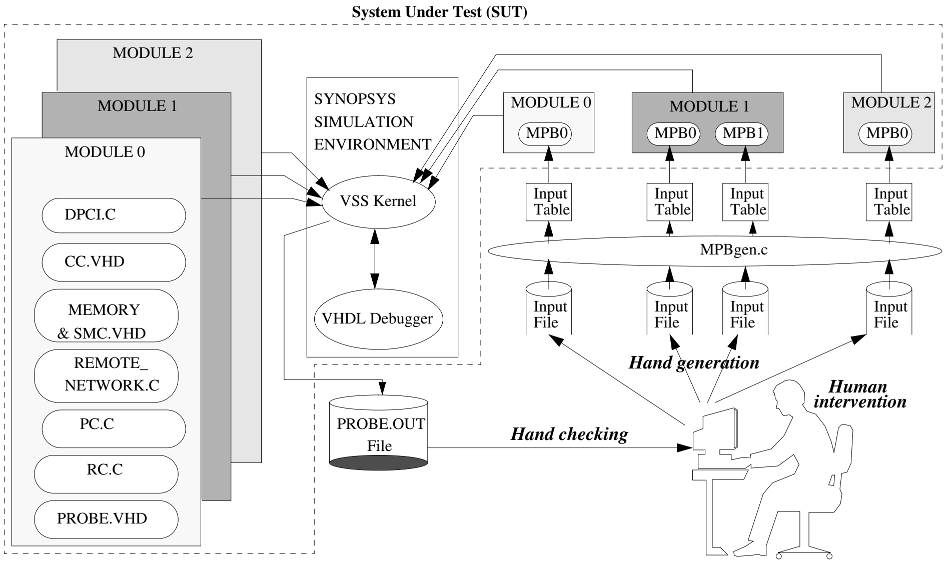

Fig. 2. The usual SIM1 testing environment

by one CPU, is achieved by using a “barrier” (the SYNC_CYC transaction) for any subsequent operation issued by the same CPU. In the situation where transactions are executed by several CPUs, the way to achieve synchronization between these CPUs consists in using an operation in which one specifies a delay δ after which each transaction (one input file to its corresponding CPU) will be executed by turns on the bus. The main difficulty of this synchronization mechanism is the estimation of δ which is currently done empirically.

The probe output files: A probe output file is generated at each clock cycle if significant events happen (see Figure 2). It contains for each module the sequence of actions which has been effectively executed in the system together with the Presence Cache and Remote Cache status. One line of this file describes one action with a stamp corresponding to the starting time of its execution and has the following form:

```
PROBE #0 ---> L_Bus 620 burst rwitm A0 Tag 00 addr=014000AA00
                                        Pos_Ack Resp_Rerun at time 660 NS
```

This line means that the probe of Module#0 observes at time `660 NS` a `RWITM` transaction on the local bus 620.

2.4 The current testing methodology

Currently, the input tables are written “by hand” and the analysis of the output file is also done “by hand” using some empirical rules. It consists in comparing each line of the probe file with what was specified in the test purpose which is informally described in the test plan document. *The main problem here is the* *analysis task which is completely based on informal specifications and informal notion of conformance.* This implies the problem of the correctness of these tests, and therefore the problem of the confidence to put in the associated verdicts. The approach using TGV brings a solution to this problem since all the objects (specification, test purposes,...) are formally specified.

3 Automatic tests generation with TGV

The prototype TGV we have developed in the Pampa team at Inria-Rennes in collaboration with the Spectre team of the Verimag laboratory at Grenoble is dedicated to automatic generation of conformance tests for protocols based on their formal specification. Given a formal specification of the system to be tested and a formal description of a test purpose (which represents an abstract form of the property to be tested), TGV generates an abstract test case. It is a direct acyclic graph in which each path represents a test sequence with associated verdict which indicates whether the implementation under test (IUT) conforms with the specification or not. Details on TGV algorithms can be found in [5–7]. We present here only the elements (described in Figure 3) which participate in the generation of a test case for the Bull's CC_NUMA architecture.

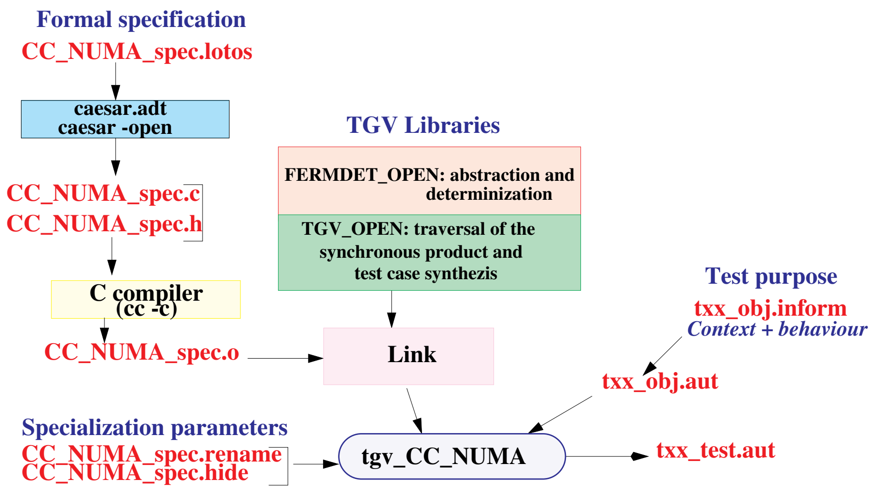

Fig. 3. TGV General Architecture using LOTOS entry

The first main entry of TGV is the formal specification of the system. The CAESAR.ADT compiler of the CADP toolbox [8] is used to compile the *data part* of the specification. The CAESAR compiler produces the C file corresponding to the *control part*, including the functions (Init, Fireable, Compare,...) needed by TGV to manipulate "on-the-fly" the state graph of the system (without generating it) [5]. Then, the C compiler produces the corresponding object-file (CC_NUMA_spec.o in Figure 3).

Depending on the properties to be tested, some observable interactions described in the LOTOS specification can be judged not important for the testing activity. Those interactions must be considered unobservable. This is done in TGV by a *hiding* mechanism (CC_NUMA_spec.hide in Figure 3) which contains all the interactions to be considered internal to the system. The semantics of LOTOS (so do the CAESAR compiler) does not make distinction between input and output because interactions between processes are synchronization events. But, TGV needs to distinguish controllable events (from tester to implementation) from observable events (from implementation to tester) in the generated test cases. We introduce in TGV a *renaming* mechanism to resolve this problem.

The other main entry of TGV is the formal test purpose from which we have to generate a test case. It is formalized (see an example of formalization in section 3.2) by an automaton in Aldebaran format. The libraries FERMDET_OPEN and TGV_OPEN contain the functions which realize "on-the-fly" all the operations (abstraction, reduction, determinization and test case synthesizing) leading to the generation of the test case. This is a solution to the combinational explosion problem which makes most of tools unable to generate test cases for complex systems. Linking the object file together with the two libraries (FERMDET_OPEN and TGV_OPEN), produces an executable (tgv_CC_NUMA in Figure 3).

Given a formal test purpose (txx_obj.aut) and the specialization files (described with two files CC_NUMA_spec.rename and CC_NUMA_spec.hide) as parameters of this executable, TGV generates the corresponding test case in form of a "decorated" DAG (Direct Acyclic Graph). Each path of this DAG represents a test sequence.

3.1 Formal specification of the cache coherency protocol

The formal specification is composed of 3 modules and consists of about 2000 Lotos lines where one half describes the control part (13 processes) and the other half defines the ADT (Abstract Data Types) part. As this formal specification is considered by TGV as the reference model of the system, it has to be strictly debugged and verified. This has been done with appropriate formal verification techniques [8]. In the following, the 3 modules are called M0, M1 and M2. Each module contains one processor called P0. There are two block addresses in the system called A0 and A1, and two data D0 and D1. These blocks are physically located in module M0. Two main reasons bring us to make some abstractions:

• The first reason is due to the size and the complexity of the BULL'S CC_NUMA architecture, with as direct consequence the state explosion problem even though TGV works "on-the-fly". Thus, some causally dependent operations concerning the same transaction are collapsed. In a remote transfer for example, an event from the sending agent is followed by an event for the receiving agent. In order to reduce the complexity, these two transactions are collapsed in one event and modeled in the Lotos specification by a gate.

• The second reason is that in this work, we are interested in tests generation for the Cache Coherency Protocol. So, we make abstractions needed to hide all other operations which do not concern with this protocol. For example the local response transaction always follows a local bus transaction in an atomic way (although if the real system can do something else between this two actions). These two transactions are collapsed in the Lotos specification, as well as all events between bus operation and response.

Notice here that these abstractions do not change the semantics, since during the execution we also do appropriate corresponding abstractions on the probe output files (see later the TRANSLATOR application in section 4).

3.2 Formalization of the test purposes

In TGV, a test purpose describes an abstract view of the test case and it is modeled by a labeled automaton in the Aldebaran syntax [8]. The format of a transition is: (from_state, label, to_state). A label is a LOTOS gate followed by a list of parameters. As an example, we give hereafter (see Figure 4) the automaton which formalizes the informal test purpose described in section 2.2.

```
des (0,8,7)
(0,"?BUS_TRANS !M1 !FLUSH !A0 !PROCESSOR !FALSE",1)
(1,"*",1)
(1,"BUS_TRANS !M0 !FLUSH !A0 !RCC_INQ !FALSE",2)
(2,"?BUS_TRANS !M0 !RWITM !A0 !PROCESSOR !FALSE",3)
(3,"LMD_GET !M0 !OUTQI0 !A0 !A0 !RCC_INV !FLAG(FALSE, FALSE) !BCK_COLL",4)
(4,"LOC_RESP !M0 !ARESP_RETRY",5)
(5,"*",5)
(5,"PACKET_TRANSFER !M0 !M1 !RESP_PACKET_TYPE !NIL_DATA !NETRESP_DONE
                                                              !OUTQI0",6)
ACCEPT 6
```

Fig. 4. An example of formalized test purpose

As said before, TGV needs to distinguish between input and output actions of the system. This is achieved simply by the first occurrence of "?" (for input) or "!" (for output) in the label. One can easily recognize the transitions corresponding to the actions described in the informal test purpose. For example, the first transition indicates that the Module#1 requests for a FLUSH transaction on the block address A0.

The statement ACCEPT 6 indicates to TGV that the state 6 is the acceptance state of the test purpose. When the Module M0 sends a response (noted NETRESP_DONE) to Module#1 which notifies the good completion of the transaction, TGV should consider that the test purpose is reached. This is mentioned in the test purpose with the last transition. The label "*" stands for *otherwise.*

With the transition (1,"*",1), TGV takes other intermediate observations into account until it observes the specified observations (from state 1).

We do not describe here the complete test purpose including the refusal state which indicates to TGV to not consider the labels of transitions which lead to that state while generating the test case. In our example, there are 36 transitions like this which are repeated for all the 5 non-final states of the test purpose. This can seem complicated but we have developed a software which automatically generates these transitions.

3.3 Generated abstract test cases

We are not going to describe all the generated test case (it contains 20 transitions and 20 states) starting from the formal test purpose described in section 3.2 corresponding to the informal one of section 2.2. Notice that most of the generated test cases contain more than 400 states and transitions. Due to their complexity, such test cases are difficult to obtain by hand even by experts. We give here and comment some of the first significant lines of the test case:

```
des (0, 20, 20)
(0,"!BUS_TRANS !M1 !FLUSH !A0 !PROCESSOR !FALSE",1)
(1,"RCT_GET ?M1 !OUTQI0 !A0 !A0 !RCC_INV !NO_COLL",2)
(2,"LOC_RESP ?M1 !ARESP_NULL",3)
(3,"PACKET_TRANSFER ?M1 !M0 !FLUSH !A0 !REQ_PACKET_TYPE !NIL_DATA
                          !NETRESP_NIL !OUTQI0 !M1 !OUTQI0 !C0",4)
(4,"LMD_GET ?M0 !INQI0 !A0 !A0 !RCC_INV !FLAG(FALSE, FALSE) !NO_COLL",5)
(5,"BUS_TRANS ?M0 !FLUSH !A0 !RCC_INQ !FALSE",6)
(6,"LOC_RESP ?M0 !ARESP_RETRY",7)
(7,"!BUS_TRANS !M0 !RWITM !A0 !PROCESSOR !FALSE",8)
(8,"LMD_GET ?M0 !OUTQI0 !A0 !A0 !RCC_INV !FLAG(FALSE, FALSE) !BCK_COLL",9)
(9,"LOC_RESP ?M0 !ARESP_RETRY",10)
(10,"BUS_TRANS ?M0 !FLUSH !A0 !RCC_INQ !FALSE",11)
(11,"LOC_RESP ?M0 !ARESP_NULL",12)
(12,"PACKET_TRANSFER ?M0 !M1 !RESP_PACKET_TYPE !NIL_DATA !NETRESP_DONE
                                                     !OUTQI0, (PASS)",13)
.....
```

In addition to other intermediate actions generated by TGV, one can recognize the reverse form (output becomes input) of actions described in the formal test purpose. Thus, the first transition is the first stimuli of the tester and consists of a FLUSH transaction requested by module M1. This is expected to be a remote operation as the target of this transaction is the address location A0 (local to M0). The transition (5,"BUS_TRANS ?M0 !FLUSH !A0 !RCC_INQ !FALSE",6) indicates that the FLUSH operation is correctly arrived on Module#0 and has been run on local bus of Module#0. So, at that point every local operations of Module#0 on the same address A0 (in the example: (7,"!BUS_TRANS !M0 !RWITM !A0 !PROCESSOR !FALSE",8)) leads to a block collision: (8,"LMD_GET ?M0 !OUTQI0 !A0 !A0 !RCC_INV !FLAG (FALSE, FALSE) !BCK_COLL",9). At that point, in conformity with the specification, the local operation is retried until the remote operation has completely accomplished: (9,"LOC_RESP ?M0 !ARESP_RETRY",10). The remote operation on module Module#0 ends with a DONE remote response on remote link: (18,"PACKET_TRANSFER ?M0 !M1 !RESP_PACKET_TYPE !NIL_DATA !NETRESP_DONE !OUTQI0, (PASS)",19). The other transitions of this test case correspond to other orders of execution of the operations previously described.

4 Making the generated test cases executable in the SIM1 environment

An abstract test case generated by TGV is a direct acyclic graph in which each branch describes a sequence of interactions between the tester and the system under test. This way of generating test cases is suitable to network protocols conformance testing where the testing activity is "interactive". Even though some tests would better be executed in an interactive way, we have seen (see section 2.3) that the usual testing activity in SIM1 environment is rather off-line ("batch") as it consists in 3 independent steps: (a) stimulating the system, (b) collecting all the observations, (c) analyzing and emitting a verdict.

Our first deal was to demonstrate that this usual manual testing approach can be done automatically. So, we have implemented a batch testing environment for the execution of interactive abstract tests. Figure 5 shows the overall structure of the tester package we have developed. It consists of three applications called EXCITATOR, TRANSLATOR and ANALYSOR. A complete example of how these applications fit together to execute batch tests is described in [9]. Notice that the main difference with the interactive approach described hereafter is that in the batch approach, the launching by turns of the 3 applications (first EXCITATOR, second TRANSLATOR and third ANALYSOR as indicated in figure 5) is done by hand and once for each test case. Moreover some tests (which needs interactivity) cannot be efficiently executed. After this, we have proposed and implemented also an interactive testing environment in order to keep the gain brought by the interactive nature of the tests generated by TGV.

4.1 The interactive version of the tester package: an example

Let us now consider the informal test purpose corresponding to an address collision situation described in section 2.2. In this case, the test case is clearly interactive because after the FLUSH transaction (requested by Module#1), the RWITM operation (requested by Module#0) can be initiated only after the observation of the FLUSH operation on the local bus of Module#0, which means that the operation is accepted by the Presence Cache of Module#0.

We are not going to describe all the steps of the interactive execution of this test case. The most important point here is to show through some significant steps how the 3 applications (EXCITATOR, TRANSLATOR and ANALYSOR) fit together for interactive testing.

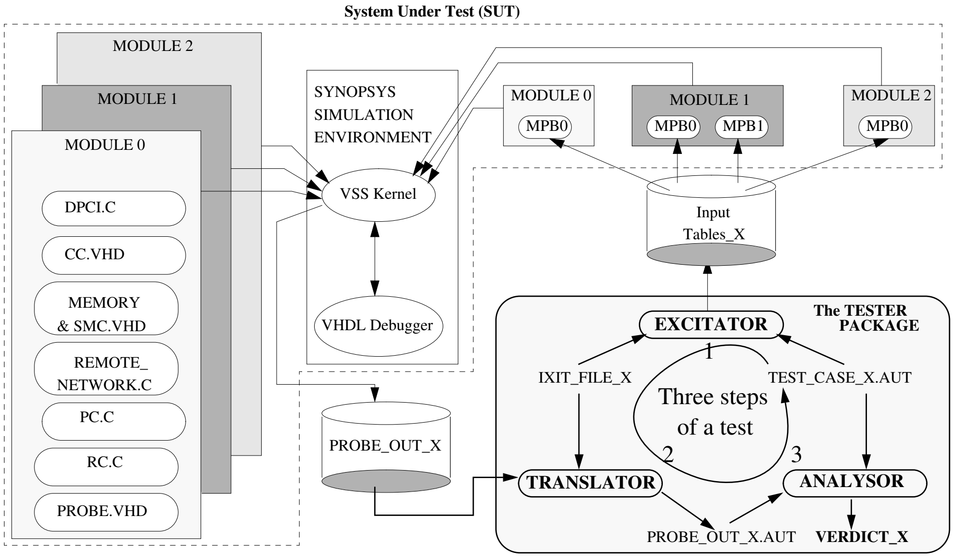

Fig. 5. The BULL'S CC_NUMA SIM1 testing environment using TGV tests

Both EXCITATOR and TRANSLATOR take into account some Implementation eXtra Informations for Testing (called IXIT_FILE_X in Figure 5). These informations describe the mapping between the abstract data values of the formal specification and the real data values of the system under test.

The **EXCITATOR** application deals with the conversion of stimuli included in the test case (called TEST_CASE_X.AUT in Figure 5) described in the Aldebaran format of TGV into a format readable by the MPBs. Once the conversion is done, the EXCITATOR proceeds to the stimulation of the MPBs. At the initial clock cycle of the simulation, EXCITATOR is invoked to extract the first stimulus from the test case (the FLUSH requested by Module#0) and proceeds to the stimulation of the MPBs. Then, the VSS kernel generates the probe output line given below (called PROBE_OUT_X in Figure 5). This line describes the requested transaction effectively observed from the system under test.

```
****** Launching the simulation:
# run 10000
*****  first excitator action: START!!
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *       MPB/620  Bus, Behavioral Model PseudoCompiler        *
    *                       Jan 29,  1998                        *
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
End of run
command detected MPB num.: 8 command : word(h0000,[]    ,z000FFC020000
0000000000010000000000000000000000,[]).
PROBE # 1 ---> L_Bus 620 16 byte flush A0 Tag 00 addr=0000000000
                                       Pos_Ack Resp_Null at time 580 NS
```

The **TRANSLATOR** application is then in charge of translating the probe outputs into a trace in the specification model. This translation is necessary to make possible the analysis of the observation according to what has been foreseen in the specification. At the next clock cycle, TRANSLATOR converts the probe output line into the Aldebaran format using appropriate abstractions corresponding to those done in the formal specification (see section 3.1). The result is submitted to the ANALYSOR application.

The obtained trace is analyzed by **ANALYSOR** according to the test case (called TEST_CASE_X.AUT) generated by TGV (see Section 3.3). Hereafter is the output of the ANALYSOR which describes the part of the test case which has been traversed.

```
------------------ANALYSER phase...
TC traversed part...
(0," BUS_TRANS !M1 !FLUSH !A0 !PROCESSOR !FALSE",2)
(2,"LOC_RESP !M1 !ARESP_NULL",3)
done
```

The probe output file generated at each next clock cycle, is converted and analyzed until one of the following conditions handles in the test case traversed synchronously by ANALYSOR:

– a verdict is found (an end of the test case is reached): it is then emitted, the following probe lines generated by the VSS kernel are ignored, and the simulation is interrupted,
– another stimulus is found: the EXCITATOR is then invoked to submit it to the VSS kernel, and a new turn of test cycle begins (simulation continues),
– no corresponding transition is found: a verdict FAIL is emitted indicating that the implementation doesn't conform to the specification (w.r.t. to the corresponding test purpose).

Let us jump now to the step of the simulation where the second stimulus (RWITM) is detected by the EXCITATOR. The collision in the home module (Module#0) is effectively obtained:

```
------------------EXCITATOR ACTION..
-- Input file for Proc. 0
RWITM WT ADDR=0000000000 TARGET=0
command detected MPB num.: 0 command : word(h0000,[]    ,z000FFC360000000
0000000010000000000000000000000,[]).
RCC   # 0 --->  out_q_behavior.vhd: L_Respout_Retry for Collision detected
                                        dbg_info= 000   at time 2240 NS
RCC   # 0 --->  out_q_entry.vhd: freeing OUT_Q # 02
                                        dbg_info= 000   at time 2260 NS
PROBE # 0 ---> BLINK SID=2 fm 0000 to part 0010 R_tag=05 R_Done
                                                        at time 2300 NS
PROBE # 0 ---> L_Bus 620  burst  rwitm     80 Tag 00 addr=0000000000
                                   Pos_Ack Resp_Retry    at time 2320 NS
```

At the last analysis phase, a PASS verdict is detected by the ANALYSOR (see below). This verdict states that the behavior of the IUT is conform to the specification w.r.t the test purpose. It means that the FLUSH operation terminates correctly, passing all the check-points despite the colliding RWITM transaction.

```
------------------ANALYSER phase...
TC traversed part...
(9,"PACKET_TRANSFER !M0 !M1 !RESP_PACKET_TYPE !NIL_DATA !NETRESP_DONE
!OUTQIO, (PASS)",1)
=>IUT(0),TC(9): ***** PASS ...
********** End of Test Case *************
```

The main difference with the batch testing is that all this steps are chained up automatically using the clock cycles and as many times as possible until the end of the test case. This allows the execution of tests in which more than one stimulus are necessary and the next stimulus depends on the reactions of the system observed after the previous stimulus, as the case with tests generated by TGV*. By the way, this approach also increases the test coverage.*

5 Results of the experiment and analysis

Through the different steps of this experiment described in the following, we indicate how we have resolved the different problems encountered, how much does it cost, what are its significant results, etc.

Formal specification The first work was to obtain a formal specification of the Bull's CC_NUMA architecture as suitable as possible for describing hardware and for test generation using TGV. The justifications of the choice of LOTOS language are given in section 3.1. In fact, good abstractions were also done in order to avoid needless complicated aspects of system in the specification (see section 3.1). As this specification is considered as a reference by TGV, it was important to guarantee that it is error-free. This work was done by Bull and took about 8 man×months to have the first version. Modifications were done until the end of the experiment. Starting from the formal specification used for verification it took 1 man×month to adapt it for test generation purpose. By the way, notice that some bugs have been detected during this formal specification.

Improvements of TGV The first version of TGV (before this experimentation) accepts only specifications in SDL or Aldebaran language. Because LOTOS language have been chosen for the specification, we were obliged to make TGV taking into account specifications described in this language. Different problems and corresponding solutions developed are explained in section 3. Other improvements of TGV dedicated to refine the generated test cases were needed and implemented during the experiment such as:

- the introduction of refusal states in the test purposes which reduce the part of the specification traversed,

– the generation of loops in the test cases (this was not the case before this experiment) leading to fewer Inconclusive verdicts; this allows the test of more functionalities.

These works were done by Inria-Rennes and costs about 8 man×months. The main benefit is that this experiment is the first one showing the interest of the on-the-fly generation available in TGV. In fact, it was impossible to obtain the state graph of the Bull's CC_NUMA specification. So, the only way to obtain tests is to work on-the-fly.

Abstract test cases generation We have formally specified all the test purposes described in the Test Groups 3 and 4 (see section 2.2) including those requiring an interactive behavior of the system. This work costs about 15 man×days. For each test purpose, we have generated the corresponding abstract test case using TGV. He who can do more can do less, we have also generated test cases for some basic operations. A total of 75 tests have been generated and cost 1 man×month. The main problem here concerns with the time cost of the test generation with TGV: from less than 1 second for some test to about 12 hours for others. This is due to the complexity of the BULL'S CC_NUMA architecture specification which required us sometimes to refine the test purposes in order to speed up the test generation with TGV.

Developing the tester package The main difficulty in executing the test cases was in the fact that the format of the test cases is different from the probe output format. It costs about 5 man×months to Inria-Rennes to develop the tester package which brings solution to this problem.

Since the applications which constitute this tester package are generic and automatically produced using classical compiler generators, they can be reused to test other systems without major effort.

Using the tester package All the test cases generated by TGV have been executed in the testing SIM1 environment using the tester package. For each test case and the corresponding probe output file, no sensible overhead is charged to the simulation time due to the presence of the tester package. An estimation of maximal time spent to execute all the 75 tests is less than 20 hours (1 day full time basis) corresponding to 1000 cycles per test, 0.6 second per cycle, 5 minutes for environment loading.

Results and analysis The main benefit in using the TGV approach is that we only have to formally specify the system to test and the test purposes, then all the testing activity would be completely automated. The time spent in specifying the BULL'S CC_NUMA architecture, formalizing test purposes and generating the test cases with TGV is completely paid by the better correctness and the confidence to put in the implementation. This approach permitted to detect 5 bugs concerning principally the address collision, and problems of test coverage (some situations were not tested): the update of the Presence Cache and Remote Cache directory sometimes are not done in the same order as described in the specification.

6 Conclusion

In this paper, we have presented an end-to-end industrial experiment which demonstrates that the prototype TGV which was developed for conformance testing of communication protocols can also be efficiently used to test hardware architectures. In fact and this is the main result of this experiment, the approach have permitted to improve the quality of the tests and the test coverage: we have detected bugs which were not detected manually by experts of hardware testing, using interactive approach. It brings also some significant improvements in both of the conformance test generation with TGV at Inria-Rennes and off-line testing in hardware at Bull: this approach will be used for another architecture under construction at Bull.

Now, we are on the way to improve again our test coverage using more general test purposes and living TGV to decide the actions to do on the system to cover a particular situation.

References

1. IEEE (Institute of Electrical and Electronic Engineers). Standard VHDL Language Reference Manual. IEEE Standard 1076-1993, 1993.
2. IEEE (Institute of Electrical and Electronic Engineers). Standard Verilog HDL Language Reference Manual. IEEE Draft Standard 1364, October 1995.
3. M. Faci and L. Logrippo. Specifying Hardware in LOTOS. In D. Agnew, L. Claesen, and R. Camposano, editors, *In the 11th International Conference on Computer Hardware Description Languages and their Applications*, pages 305–312, Ottawa, Ontario, Canada, April 1993.
4. G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian. Specification and Verification of the PowerScale^{TM} Bus Arbitration Protocol : An Industrial Experiment with LOTOS. In R. Gotzhein and J. Bredereke, editors, *Proceedings of FORTE/PSTV'96*, Kaiserslautern, Germany, October 1996.
5. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In A. Alur and T. Henzinger, editors, *Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA*, LNCS 1102. Springer, July 1996.
6. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Journal of Science of Computer Programming - Special Issue on Industrial Relevant Applications of Formal Analysis Techniques*, 29, p. 123-146, 1997.
7. M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, L. Mounier, and P. Morel. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming - Special Issue on The Application of Formal Methods in Industrial Critical Systems*, To appear, 1999.
8. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, LNCS vol. 1384, p. 68-84, March 1998.
9. H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In A. Petrenko, and N. Yevtushenko, editors, *IFIP TC6 11th International Workshop on Testing of Communicating Systems*. Chapman & Hall, p. 211-226, September 1998.