An industrial experiment in automatic generation of executable test suites for a cache coherency protocol*

Hakim Kahlouche§, César Viho¶, Massimo Zendri‖
E-mail: {kahlouch, viho}@irisa.fr, massimo.zendri@bull.net

March 13, 1998

Abstract

In this paper, we present an end to end industrial case-study concerning the automatic generation of tests suites for the Cache Coherency Protocol of a Multiprocessor Architecture. It consists of the following stages : (1) formal specification of the architecture using Lotos language, (2) formal description of the test purposes, (3) automatic generation of abstract test suites using the prototype TGV, and (4) automatic generation and analysis of executable test suites.

Through the description of each of the previous stages, this paper demonstrates that tools designed for protocol conformance testing can be efficiently used to generate executable tests for hardware concurrent systems.

Key-words : Conformance Testing, Test Generation, Lotos, Test Execution, Hardware Multi-Processor Architecture, Cache Coherency Protocol.

1 Introduction

The aim of testing is to guarantee that the implementation of the system correctly realizes what is described in its specification. In this paper, we are particularly concerned with the so called *black box conformance testing*. In this testing approach, the behaviour of the implementation (otherwise called IUT for Implementation Under test) is known only by its interactions with the environment via its interfaces (called PCO for Point of Control and Observation). Thus, testing consists in stimulating the IUT and observing its reactions on its PCOs. The prototype TGV has been developped to generate test suites for communication protocols using the black box conformance testing approach. TGV is based on protocol verification algorithms and its main purpose is to fit as well as possible the industrial practice of test generation. It takes as entries the formal specification of the system to be tested and a formalization of a test purpose, and it generates an abstract test case. A test case is represented by a tree (not only sequences), and each branch of this tree contains the interactions between the tester and the implementation. A verdict is associated to each branch. TGV has been experimented on the Drex military protocol [1]. The formal specification of this protocol is done in SDL formal description language (IUT-T Recommendation Z.100). The comparison of the hand written test cases with those generated by TGV, has showed the interest and efficiency of TGV [2].

*This work has been done in the context of Dyade (R&D joint venture between Bull and Inria).

§INRIA-Rennes, Campus universitaire de Beaulieu, F35042 Rennes, France
¶IRISA-IFSIC/Université de Rennes I, Campus de Beaulieu, F35042 Rennes, France
‖Bull Italia, Via ai Laboratori Olivetti, I-20010 Pregnana Milanese (MI), Italy

On another side, many tools have been developped in the area of Hardware testing, allowing simulation of the specifications, automatic synthezis of implementations and even tests generation. The hardware design is often based on hardware description languages such as VHDL (IEEE Standard 1076-1993). This is due to the possibility in these languages to describe hardware-related details such as register-transfer, gate and switch level. This details may lead in overspecification and (most of all) are not directly relevant to high-level functionalities specification, such as Cache Coherency Protocols, etc. It then becomes fully justified to wonder whether the formal specification languages and associated tools designed in computer network area could be better appropriate for the description of these functionalities [3, 4].
The challenge for us, in the VASY (VAlidation of SYstems) action of the Dyade GIE Bull/Inria, is to demonstrate that TGV can also be used to generate tests for other systems than communication protocols: particularly, for the Cache Coherency Protocol of a Multiprocessor Architecture under construction at Bull Italia. In this experiment, we have had to deal with three main difficulties:

- The system to test is not a communication protocol but a Cache Coherency Protocol of a Hardware Multiprocessor Architecture.
- The formal description language used is not SDL but LOTOS (ISO International Standard 8807).
- The habits and methodologies of test practicians in the area of hardware architecture are not the same as in the area of communication protocol testing.

First, we describe the fundamental aspects of TGV. We give some precisions on the Cache Coherency Protocol of the architecture[1]. In this paper, we will call this architecture the BULL'S CC_NUMA Architecture. Then, we indicate the appropriate abstractions made on its LOTOS formal specification, in order to make the test generation feasible. We show how we have used this formal specification to generate tests suites with TGV, particularly how the test purposes have been formalized. We have also developped tools in order to make the abstract test cases generated by TGV executable in real test environment of the BULL'S CC_NUMA Architecture. The last section is dedicated to the presentation of these tools. We end this paper by reporting results of the experiment which indicates how we have resolved the main difficulties enumerated above.

2 Overview of tests generation with TGV

In this section, we state precisely the context of black box conformance testing of TGV. Then, we give the principles of automatic test suites generation based on the formal specification of the component to be tested and test purposes.

2.1 Context of TGV

The testing method with TGV consists in stimulating the IUT and observing its reactions on its PCOs. Depending on what is observed, a *verdict* is emited indicating whether the IUT can be considered as a good implementation of the system or not. According to the importance of this verdicts, it is fundamental to give a precise specification of the system. It is also important to define the *conformance relation* between an implementation and the specification. Because a test activity

[1] For confidential reasons, we cannot give precise details on this architecture

of complex system cannot be exhaustive, only particularly important aspects can be tested. This can be done by defining *test purposes* which help to choose the behaviours of the IUT to be tested. In the testing methods where the test suites are hand written, all the objects enumerated before (specification, test purposes, conformance relation, verdicts) are described informally. This implies the problem of the correctness of these test suites, and therefore the problem of the confidence to put in the associated verdicts. The methods of automatic generation of test suites which are based on *formal description* of these different objects bring a solution to this problem. The prototype TGV we have developped in collaboration with the Spectre team of the Verimag laboratory [1, 5] is precisely situated in this context.

2.2 Principles of automatic generation in TGV using LOTOS specification

TGV takes two main entries : the formal specification of the system and a formal description of a test purpose (by an automaton) which represents an abstract form of the test case to be generated. From these objects, TGV gives as result a test case in form of a "decorated" DAG (Direct Acyclic Graph). The paths of this DAG (which can be unfolded into a tree) represent test sequences. In what follows, we summarize the operations done by TGV to generate this DAG (details can be found in [5, 6]):

Abstraction, Reduction and Determinization This consists in abstracting the state graph of the system according to observable events through its interfaces with the environment. This operation generates *internal actions* which are not needed in black box testing. Thus, TGV realizes a reduction operation (the $\tau^* a$-minimization) to suppress internal actions and bring out only the observable behaviour of the system. The non-determinism thus introduced by the previous operation is suppressed by a determinization operation.

Test Case Synthesizing The TGV algorithm is based on a depth first search traversal of a synchronous product of the abstracted state graph obtained (as shown above) and the formal test purpose. During the traversal, we verify that the test purpose is coherent with the specification. While backtracking, we synthesize a "decorated" DAG (called the *test case*) which consists of *preambule, test body, postambule, verdicts*. This graph satisfies the *controlability condition* which stipulates that the tester controls its outputs.

The test graph integrates also *timers management* which guarantees the termination of the application of the tests on implementations.

We present here the elements (described in Figure 1) which partake in the generation of a test cases for the BULL'S CC_NUMA architecture.
The first main entry of TGV is the formal specification of the BULL'S CC_NUMA machine. The LOTOS language has been selected because its underlying semantics model is based on the rendezvous synchronization mechanism which is well suited for the specification of hardware entities [7, 3, 4] such as processors, memory controllers, bus arbiter. To the environment point of view, this entry (CC_NUMA_spec.lotos in Figure 1) defines the temporal relation among the interactions that constitute the externally observable behaviour of the system to be tested. This specification has been debugged and verified with appropriate formal verification techniques, and is considered by TGV as the reference model of the system.

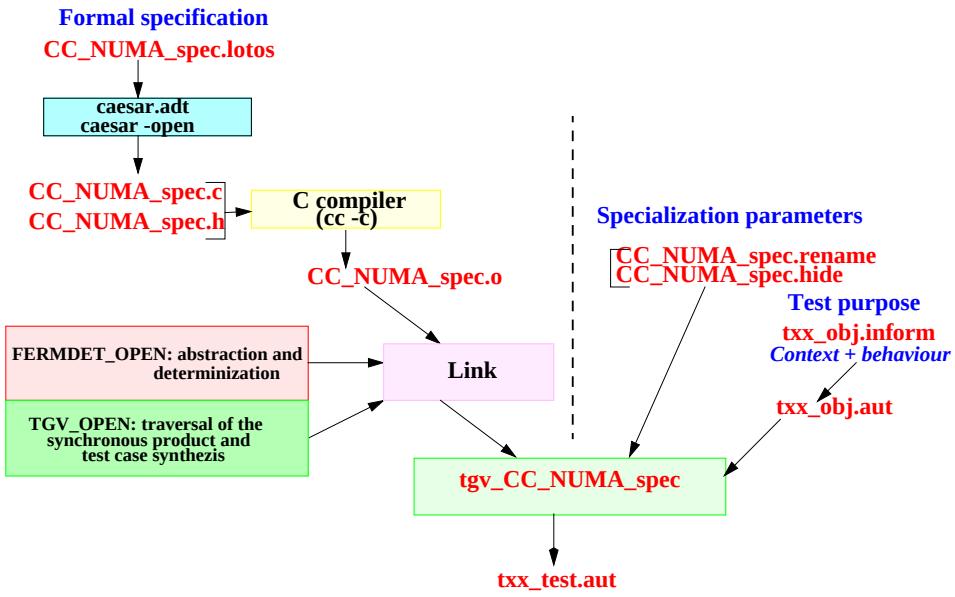

Figure 1: TGV General Architecture using LOTOS entry

The CAESAR.ADT compiler of the CADP toolbox [8, 9] 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) [6]. Then, the C compiler produces the corresponding objectfile (CC_NUMA_spec.o in Figure 1).
Some observable interactions described in the LOTOS specification can be judged not important to the test activity point of view. Those interactions must be considered unobservable. This is done in TGV by a *hiding* mechanism (CC_NUMA_spec.hide on the figure 1) which contains all the interactions to be considered internal to the system.
The semantics of LOTOS (so do the CAESAR compiler also) does not make distinction between input and output. In fact, interactions between processes are synchronization events. This puts in trouble TGV in which this disctinction is needed to distinguish controlable events (from tester to implementation) from observable events (from implementation to tester) in the generated test cases. We introduce the *renaming* mechanism in TGV to resolve this problem.
The other main entry of TGV is the formal test purpose from (and for) which we have to generate a test case. It is formalized by an automata in Aldebaran format (see an example below in section 4.2.2), which represents an abstract view of interactions between the tester and the implementation on which the generated test case has to focus.
The libraries FERMDET_OPEN and TGV_OPEN of TGV 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 combinatory explosion problem which makes most of tools unable to generate test cases for complex system. In another side, this makes algorithms of TGV more complicated but *offers the possibility to generate test cases even when the state graph of the system to be tested cannot be generated*, as it is the case in the experiment we are describing in this paper.
Linking the object file together with the two libraries (FERMDET_OPEN and TGV_OPEN), produces an executable (tgv_CC_NUMA_spec in Figure 1). Given a formal test purpose (txx_obj.aut) and

a test architecture (described with two files CC_NUMA_spec.rename and CC_NUMA_spec.hide) as parameters of this executable, TGV generates the corresponding test case.

3 The BULL'S CC_NUMA architecture: 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), derived from Stanford's DASH multiprocessor machine [10]. It consists of a scaleable interconnection of modules. The memory is distributed among the different modules. Each module contains a set of processors (see figure 2).

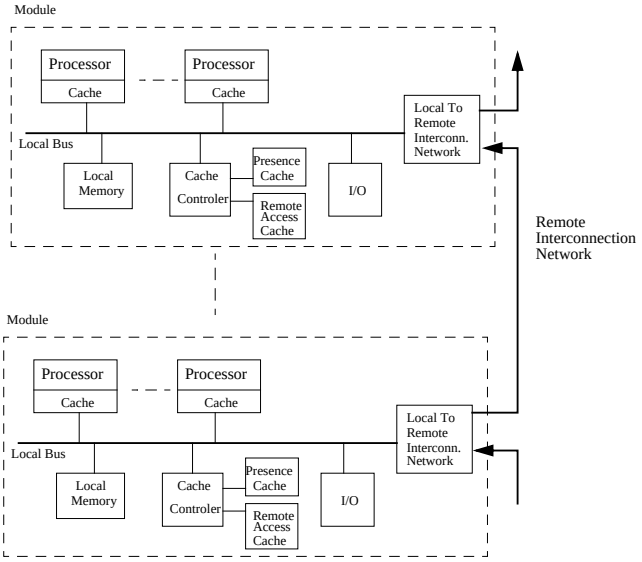

Figure 2: The BULL'S CC_NUMA General Architecture

The BULL'S CC_NUMA architecture key feature 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 (and only those) 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 (whose home is in a module different from the local module). 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.*
So, the purpose of testing the Cache Coherency Protocol consists in ensuring that the status of the Presence Cache and Remote Cache are allways correctly updated during the execution of any transaction in the BULL'S CC_NUMA architecture.

4 Formalization and abstract tests generation

In this section, we describe the formal objects which constitute the main entries of TGV : the formal specification of the cache coherency protocol of the BULL'S CC_NUMA architecture using LOTOS

and the formalization of the test purposes.

4.1 Formal specification of the cache coherency protocol

The LOTOS language has been selected 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 [3, 4] such as processors, memory controllers, bus arbiter, etc. The communications between these components by sending electrical signals on conductors are better described by interactions between LOTOS processes, rather than infinite FIFO queues as in SDL.
The formal specification consists of about 2000 Lotos lines where 1000 lines describe the control part (13 processes) and the other half defines the ADT (Abstract Data Types) part. This specification is composed of two modules and has been debugged and verified with appropriate formal verification techniques, and is considered by TGV as the reference model of the system. In the following, we will call these modules M0 and M1. Each module contains one processor called P0. There is two block addresses in the system called A0 and A1, and two data D0 and D1. These blocks are pysically located in module M0.
Two main reasons bring us to make some abstractions in the formal specification:

- The first reason is due to the size and the complexity of the BULL'S CC_NUMA architecture, with as direct consequence the combinatory explosion problem even though TGV works "on-the-fly".

  Thus, some causally dependant operations concerning the same transaction are collapsed. For example, from the testing point of view, 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. This reduces the complexity of the specification.

- 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.

4.2 Formalization of the test purposes

A test purpose in TGV is described by an automaton which represents an abstract view of the test case. So, in order to make TGV working, we have had to formalize each test purpose. The main test purposes to be applied to the BULL'S CC_NUMA architecture are informally described (in the shape of tables with comments) in the test plan. Seven *Test Groups* have been identified. In the experiment we are reporting in this paper, we are interested by two Test Groups (the Test Groups 3 and 4) concerning the test of the Cache Coherency protocol. In what follows, we describe more precisely these two groups.

4.2.1 The Cache Coherency Test Groups

Some other definitions are needed to make what follows more easier to understand. The *Requesting Processor* is the processor that initiates the transaction. The *Requestor* is the module that includes the Requesting Processor. The *Home Module* is the module which physically locates the requested block. The *Owner Module* is the module which hold the most updated copy of the memory block.

The *Participant Modules* are modules which are requested by the Presence Cache to participate in the cache coherency protocol.
The test purposes described in the Test Group 3 are dedicated to Cache Coherency Testing (No Participants). This means that they aim to test interactions between two modules (the Requestor and the Home) which do not need interventions of other modules. For example, the table 1 describes an informal test purpose which means: "The block address is in Module#0. The CPU#0 of Module#1 executes a READ on this address. Verify that the Presence Cache (PC) status of Module#0 changes from Invalid to Shared." In this case, we can notice that the other modules (Module#2 and Module#3) are not concerned.

Cache Coherency Tests : set-PC-to-SH					Test group #3
Operation	Parameters	Source	Target	PC	Notes
READ	-	Module#1 CPU#0	Module#0	SH	PC Status of Module#0 changes from Invalid to Shared

Table 1: Presence Cache Status Setting to (SH)ared

The test purposes described in the Test Group 4 are dedicated to Cache Coherency Testing (With Participants) in which other modules than the Owner and the Requestor are requested to realize the transaction.

4.2.2 Formal specification of test purposes

A test purpose is described with a labelled 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 said previously, 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. The automaton corresponding to a test purpose describes a point of view of the system. As an example, we give hereafter the automaton which formalizes to the test purpose described in Table 1:

```
des␣(0,␣11,␣4)
(0,"?BUS_TRANS␣!M1␣!READ␣!A0␣!PROCESSOR␣!FALSE",1)
(1,"?BUS_TRANS␣!M0␣!READ␣!A0␣!PROCESSOR␣!FALSE",2)
(1,"?BUS_TRANS␣!M0␣!READ␣!A1␣!PROCESSOR␣!FALSE",2)
(1,"?BUS_TRANS␣!M1␣!READ␣!A0␣!PROCESSOR␣!FALSE",2)
(1,"?BUS_TRANS␣!M1␣!READ␣!A1␣!PROCESSOR␣!FALSE",2)
(1,"?BUS_TRANS␣!M0␣!RWITM␣!A0␣!PROCESSOR␣!FALSE",2)
(1,"?BUS_TRANS␣!M0␣!RWITM␣!A1␣!PROCESSOR␣!FALSE",2)
(1,"?BUS_TRANS␣!M1␣!RWITM␣!A0␣!PROCESSOR␣!FALSE",2)
(1,"?BUS_TRANS␣!M1␣!RWITM␣!A1␣!PROCESSOR␣!FALSE",2)
(1,"LMD_PUT␣!M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE)",3)
(1,"*",1)
ACCEPT␣3
REFUSE␣2
```

The first line is the automata descriptor. It indicates that the first state is 0, there are 11 transitions and 4 states. The first transition indicates that the processor P0 of M1 requests for a READ transaction on the block address A0.

The statement REFUSE 2 on the last line indicates to TGV that the state 2 is a refusal state of the test purpose. The label of transitions which lead to a refusal state are not considered by TGV while generating the test case. After the READ transaction requested by M1 on A0, we don't want to consider other READ transactions. This is expressed by the 8 transitions following the first one.
The statement ACCEPT 3 indicates to TGV that the state 3 is the acceptance state of the test purpose. When the LMD status of Module M0 changes from Invalid to Shared, TGV should consider that the test purpose is reached. This is mentioned in the test purpose with the transition (1,"LMD_PUT !M0 !A0 !RCC_SH !FLAG (FALSE, TRUE)",3).
The label "*" stands for *otherwise*. This mentions to TGV what to do with other observations which are not specified in the current state (state 1) of the test purpose. With the transition (1,"*",1), we indicate TGV to take other intermediate observations into account until it observes the specified observations (from state 1).

4.3 Generated abstract test cases

In this section, we give the test case generated by TGV starting from the test purpose of Table 1 and formally described above. Then we give some explanations of its content.

```
des␣(0,␣28,␣26)
(0,"!BUS_TRANS␣!M1␣!READ␣!A0␣!PROCESSOR␣!FALSE",1)
(1,"LOC_RESP␣?M1␣!ARESP_MODIF",2)
(2,"PACKET_TRANSFER␣?M1␣!M0␣!READ␣!A0␣!REQ_PACKET_TYPE␣!NIL_DATA␣!NETRESP_NIL
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣!OUTQIO␣!M1␣!OUTQIO",3)
(3,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",4)
(3,"BUS_TRANS␣?M0␣!READ␣!A0␣!RCC_INQ␣!FALSE",5)
(5,"LOC_RESP␣?M0␣!ARESP_RETRY",6)
(6,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",7)
(6,"BUS_TRANS␣?M0␣!READ␣!A0␣!RCC_INQ␣!FALSE",8)
(8,"LOC_RESP␣?M0␣!ARESP_RETRY,␣INCONCLUSIVE",9)
(8,"LOC_RESP␣?M0␣!ARESP_NULL",10)
(10,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",11)
(10,"LOC_DATA_BUS_TRANS␣?M0␣!READ␣!A0␣!D0␣!SMC␣!RCC_INQ",12)
(12,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",13)
(12,"PACKET_TRANSFER␣?M0␣!M1␣!RESP_DATA_PACKET_TYPE␣!D0␣!NETRESP_DONE␣!OUTQIO"␣14)
(14,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",15)
(14,"RCT_PUT␣?M1␣!A0␣!RCC_SH",16)
(16,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",17)
(16,"LOC_DATA_BUS_TRANS␣?M1␣!READ␣!A0␣!D0␣!RCC_OUTQ␣!PROCESSOR",18)
(18,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",19)
(18,"FREE_OUTQ␣?M1",20)
(20,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",21)
(14,"LOC_DATA_BUS_TRANS␣?M1␣!READ␣!A0␣!D0␣!RCC_OUTQ␣!PROCESSOR",22)
(22,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",23)
(22,"RCT_PUT␣?M1␣!A0␣!RCC_SH",24)
(24,"LMD_PUT␣?M0␣!A0␣!RCC_SH␣!FLAG␣(FALSE,␣TRUE),␣(PASS)",25)
(24,"FREE_OUTQ␣?M1",20)
(5,"LOC_RESP␣?M0␣!ARESP_NULL",10)
(1,"LOC_RESP␣?M1␣!ARESP_RETRY,␣INCONCLUSIVE",9)
```

We can recognize the reverse form ("!" rather than "?", because it is the tester's point of view) of the first transition of the test purpose described before: (0,"!BUS_TRANS !M1 !READ !A0

!PROCESSOR !FALSE",1). This is a stimuli of the tester. It consists of a READ transaction on the local bus of module M1. The target of this transaction is the address location A0 (local to M0). So, this is expected to be a remote operation.
Let us now describe some important transitions of the test case :

- The transition (1,"LOC_RESP ?M1 !ARESP_MODIF",2) indicates that the remote cache controller recognizes the address as a non-local address and force the arbiter of the node to give a modify (ARESP_MODIF) response. Normally, the memory controller will do nothing, the remote cache controller is in charge of providing the data. The requesting processor waits for the data on a dedicated data-path of the cross-bar switch.

- The remote cache controller routes the request to the remote link, the request is directed to the home module M0 : (2,"PACKET_TRANSFER ?M1 !M0 !READ !A0 !REQ_PACKET_TYPE !NIL_DATA !NETRESP_NIL !OUTQI0 !M1 !OUTQI0",3). The packet contains a request to provide data to module M1.

- (3,"LMD_PUT ?M0 !A0 !RCC_SH !FLAG (FALSE,TRUE),(PASS)",4) :

  Module M0 reads his Presence Cache to see the status of the cache line. After that, the entry is updated (the line is in a Shared (RCC_SH) status), which means that the line is present also in module M1; an array of booleans is used to represent the presence bits (FLAG (FALSE, TRUE)). The verdict (PASS) is then emited indicating that the test purpose is reached.

- The implementation is allowed to choose the order in which the operations are done. Thus, the three following transitions constitute an other way to reach the test purpose : (3,"BUS_TRANS ?M0 !READ !A0 !RCC_INQ !FALSE",5), the remote cache controller of module M0 requests localy the data. An agent on the bus can always decide to retry the transaction for any sort of reason (5,"LOC_RESP ?M0 !ARESP_RETRY",6). This means that the remote cache controller has to execute again the transaction. Then the Presence Cache changes to Shared : (6,"LMD_PUT ?M0 !A0 !RCC_SH !FLAG (FALSE, TRUE), (PASS)",7).

- (6,"BUS_TRANS ?M0 !READ !A0 !RCC_INQ !FALSE",8) and (8,"LOC_RESP ?M0 !ARESP_RETRY, INCONCLUSIVE",9) indicate that the remote cache controller of module M0 executes again the local operation. A second retry response should lead to an inconclusive verdict because in TGV we have choosen to cut cycles in order to generate finite test cases. This is also the case for the transition (1,"LOC_RESP ?M1 !ARESP_RETRY, INCONCLUSIVE",9). An improvement in TGV will overhelm this problem allowing the number of retry needed.

- All the other transitions of this test case can be easily interpreted as they correspond to different other orders of execution of operations described in previous items.

4.4 Results on abstract test cases generation

At this moment of the experimentation, we have formally specified half of test purposes described in the Test Groups 3 and 4 (see section 4.2.1). For each test purpose, we have generated the corresponding abstract test case using TGV. The main problem here concerns the time spent by TGV to generate test cases. 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.

5 Implementation of the generated test cases

The purpose of this section is to describe the techniques and tools we have developed in order to make the abstract test cases generated by TGV executable in the testing environment of BULL's CC_NUMA architecture (called SIM1 environment). To do so, we start by describing the SIM1 environment and principally the testing methodology currently used in SIM1 environment. Then, we present the new testing architecture. Finally, we describe an example on using this architecture.

5.1 The current testing architecture

The SIM1 BULL's CC_NUMA testing environment structure consists of three modules, connected on a Remote Interconnection Network. Each module is composed by Processor Behavioral Models (MPB Bus Model), Memory Array and Memory Control, Arbiter and I/O Block, Coherency Controller, Remote Cache Tag that contains the Tag of Remote Cache and the Presence Cache.
Figure 3 shows the current testing environment. For sake of clarity, the MBPs in th figure are put outside of the other components of a module, even though in the environment all the components of a module are really together.
The simulation environment is based on the Synopsys simulation environment, composed of kernel event simulator (VSS kernel : VHDL System Simulator) and a front end human interface (VHDL Debugger). The models of BULL's CC_NUMA architecture are linked to the VSS in order to obtain a single executable simulator file. The VSS is in charge of downloading the outputs which are issued by the PROBE lines into a file (PROBE.OUT file in Figure 3). A CPU#i (MPBi in Figure 3) expects an *input table* which contains the commands to the MPB model in an intermediate format. The MPBgen application is in charge of converting the MPB input commands format (*input files*) into the intermediate format (*input tables*). The first step in the testing methodology is to write the input files.

5.1.1 Input files

An input file describes a sequence of transactions to be executed by one CPU. The input files are written according to the informal test purposes specified in the test plan document. This is currently done by hand. There is one input file per MPB, and the main difficulty in describing these files is the synchronization of the CPUs w.r.t the test purpose. In fact, there are two cases of synchronization:

Case 1 (Intra-CPU Synchronization): In the case where all the transactions have to be executed by one CPU, that is materialized by only one input file, the synchronization is achieved by using the SYNC_CYC transaction. This transaction is a "barrier" for any subsequent operation issued by the same processor, and this barrier, being in the processor itself, is a "global barrier" for all the possible target we can have in CC-NUMA architecture.

Case 2 (Inter-CPUs Synchronization): In the case where the transactions of the test purpose have to be executed by several CPUs, the problem is to achieve an inter-CPUs synchronization. The participant CPUs may belong to different modules. The previously described SYNC_CYC transaction can also be used in this case. Another way to achieve this synchronization consists in submitting one input file to its corresponding CPU. Then after an estimated delay δ of the execution, the next input file is submitted to an other CPU which possibly belongs to an other module. The difficulty of this synchronization mechanism lies in the estimation of δ.

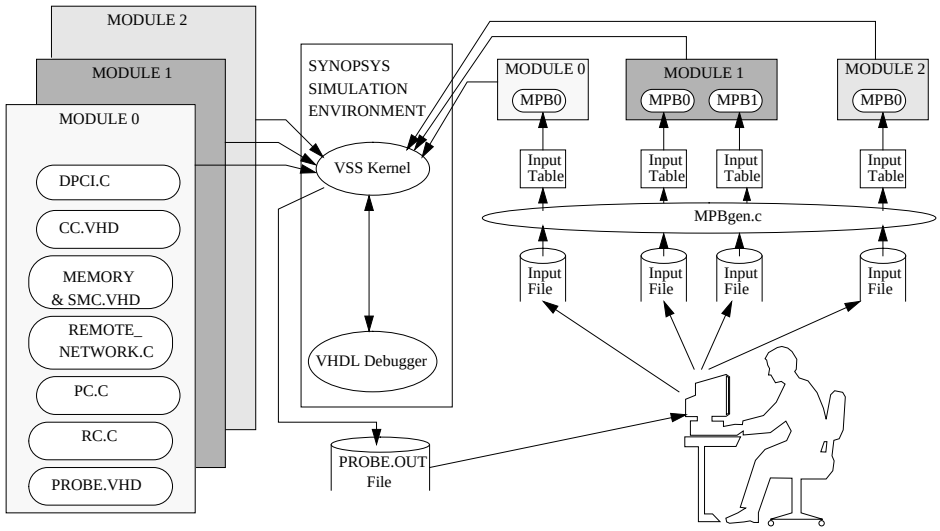

Figure 3: Current testing environment of the BULL's CC_NUMA architecture

5.1.2 Output Analysis

Once the execution of the different input files has been completed, a PROBE output file is generated (see Figure 3). This file contains for each module the sequence of actions which has been effectively executed in the system together with the Local Memory Directory and Remote Cache status. Each action is associated with a stamp, that is the starting time of its execution. One line of this file has the following form :

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

It means that the PROBE of Module#0 observes at time `660 NS` a `RWITM` transaction on the local bus 620, the tag of the transaction is `00`, the target of the transaction is the address `14000AA00`, the transaction is positively acknowledged and a `Resp_Rerun` response is locally delivered to the requestor. Currently, the analysis of the output file is 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 and what is informally described in the test plan document. Finally a verdict is emitted at the end of the analysis.
The main problem here is the analysis task which is completely based on informal specifications and informal notion of conformance which may sometimes lead to false verdicts. The TGV approach brings a solution to that problem since the verdicts are formally specified in the test cases.

5.2 The new testing architecture

The test cases generated by TGV are abstract in the sense that they are specified independently of the testing environment. There is one test case per test purpose (correponding to one of the tables described in the test plan document). In this section, we present the tools we have developped to make this abstract test cases executable in 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 means that input

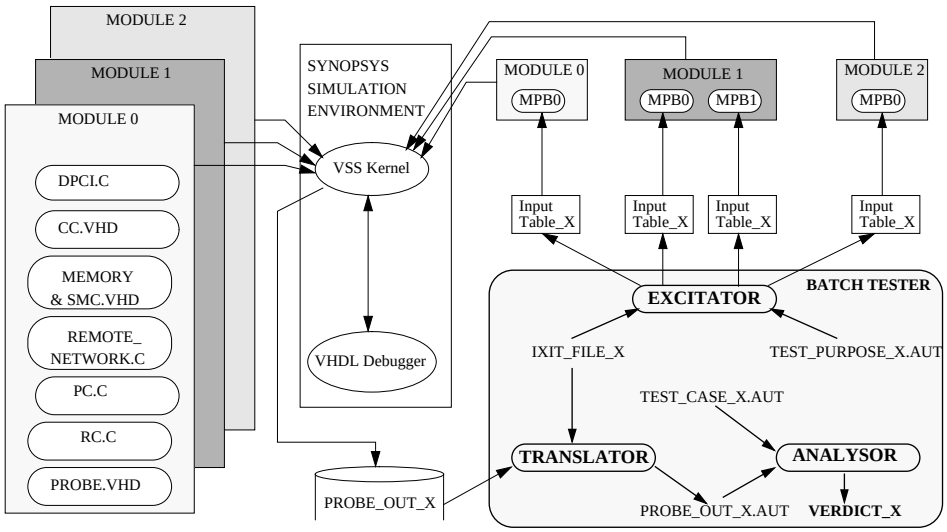

Figure 4: The BULL's CC_NUMA architecture SIM1 batch testing environment proposal

actions targeting the system under test and output actions observed by the tester are put together in the same test case. Furthermore, input actions mostly depend on the output previously observed. This way of generating test cases is suitable to network protocols conformance testing where the testing activity is "interactive". We have seen in section 5.1 that the testing activity currently used in SIM1 environment is rather "batch". Indeed, it consists in three independant steps: (a) stimulating the system, (b) collecting all what has been observed (including the stimulus), and (c) analysing and concluding with a verdict. So, the problem we have to tackle here is to implement interactive abstract test cases on top of batch testing environment. Basically, the solution we propose consists first in translating what has been observed during step (b) from the system into a trace in the model of the system. Then, this trace is analysed according to what has been forseen in the abstract test case.

Figure 4 shows the new testing environment and principally the overall structure of the batch tester package we have developed. The tester package consists of three applications.

The **EXCITATOR application** deals with the conversion of a test purpose (called TEST_PURPOSE_X.AUT in Figure) described in the format of TGV into a format readable by the MPBs. Once the conversion is done, the EXCITATOR proceeds to the stimulation of the MPBs. Then, the VSS kernel generates the probe output file (called PROBE_OUT_X). This file describes what has been effectively observed from the system under test.

The **TRANSLATOR application** is in charge of translating the probe output file into a trace in the specification model. This translation is necessary to make possible the analysis of the observation according to what has been forseen in the specification.

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

Finally, the **ANALYSOR application** proceeds to the analysis of the trace generated by the TRANSLATOR according to the given test case (called TEST_CASE_X.AUT) and delivers a verdict together with some diagnostic information. A correct trace must be a branch of the test case which

leads to a PASS verdict. The analysis consists then in sychronously traversing the trace and the test case, where the synchronization is the equality of labels (without verdict). The ANALYSOR is independant from the specification and the implementation. It only assumes that the test case and the implementation trace are described by deterministic automata.
Since the TRANSLATOR and the EXCITATOR are automatically produced using compiler generators, the tester package can be reused to test other batch systems without major effort.
In its current version, the tester package doesn't include the EXCITATOR application. Indeed, this application is quite easy to do by hand in the case of batch testing.

5.3 Example on using the tester package

We present in this section the results obtained using the tester package for the example of section 4.3.

Probe output file generation When the system under test is stimulated by the READ transaction, the VSS kernel produces the following trace. This trace describes all the operations performed by the system.

```
PROBE # 1 ---> L_Bus 620  burst  read_tt   A0 Tag 00 addr=0140042000 Pos_Ack     Resp_Mod
  PROBE # 1 ---> BLINK SID=2 fm 0010 to home 0000 R_tag=07                (rq=0010 07)  burst  read_tt
    WIM=101 addr=000140042000 retry=00 at time 880 NS
  PROBE # 0 ---> BLINK SID=0 fm 0010 to home 0000 R_tag=07                (rq=0010 07)  burst  read_tt
    WIM=101 addr=000140042000 retry=00 at time 1200 NS
  PROBE # 0 ---> L_Bus LMD update :               address = 0140042000 status =  SHD  1,
  PROBE # 0 ---> L_Bus LMD update :               address = 0000042000 status =  INV ----------------
  PROBE # 0 ---> L_Bus LMD update :               address = 0000042000 status =  INV ----------------
  PROBE # 0 ---> L_Bus LMD update :               address = 0000042000 status =  INV ----------------
  PROBE # 0 ---> L_Bus RCC  burst  read_tt   A0 Tag 7C addr=0140042000 Pos_Ack     Resp_Null
  PROBE # 0 ---> L_Bus     Data Transaction for Tag 'L111'C data=DEADBEEFFEDCBA98 at time 2260 NS
  PROBE # 0 ---> BLINK SID=2 fm 0001 to part 0011 R_tag=07 R_Done         data=DEADBEEFFEDCBA98
  PROBE # 1 ---> BLINK SID=1 fm 0001 to part 0011 R_tag=07 R_Done         data=DEADBEEFFEDCBA98
  PROBE # 1 ---> L_Bus RCT update :               address = 0140042000 status =  SHD
  PROBE # 1 ---> L_Bus     Intv. Data Xact. for Tag 00 data=DEADBEEFFEDCBA98
```

IXIT information The trace of the system is therefore submited to the TRANSLATOR application with the following IXIT information. These information give the correspondance between abstract values and real values.

```
M0 = MODULE 0
M1 = MODULE 1
A0 = ADDRESS 0140042000
D0 = DATA DEADBEEFFEDCBA98
```

Trace of the specification The TRANSLATOR application translates the trace of the system into a trace of the specification, the result is given as follow.

```
des(0,10,11)
(0,"BUS_TRANS !M1 !READ !A0 !PROCESSOR !FALSE",1)
(1,"LOC_RESP !M1 !ARESP_MODIF",2)
(2,"PACKET_TRANSFER !M1 !M0 !READ !A0 !REQ_PACKET_TYPE !NIL_DATA !NETRESP_NIL !OUTQI0 !M1 !OUTQI0",3)
(3,"LMD_PUT !M0 !A0 !RCC_SH !FLAG( FALSE, TRUE)",4)
(4,"BUS_TRANS !M0 !READ !A0 !RCC !FALSE",5)
(5,"LOC_RESP !M0 !ARESP_NULL",6)
```

```
(6,"LOC_DATA_BUS_TRANS !M0 !D0 !RCC",7)
(7,"PACKET_TRANSFER !M0 !M1 !RESP_DATA_PACKET_TYPE !D0 !NETRESP_DONE !OUTQI0",8)
(8,"RCT_PUT !M1 !A0 !RCC_SH",9)
(9,"LOC_DATA_BUS_TRANS !M1 !D0 !PROCESSOR",10)
```

Trace analysis Finnaly, the obtained trace is analysed according to the test case generated by TGV (see Section 4.3). This is done by the ANALYSOR application. The output of the ANALYSOR given below describes the traversed part of the test case during the analysis and the verdict which has been found. The pass verdict means that the system under test is conform to the specification w.r.t the given test purpose.

```
TC traversed part...
(0,"BUS_TRANS !M1 !READ !A0 !PROCESSOR !FALSE",1)
(1,"LOC_RESP !M1 !ARESP_MODIF",2)
(2,"PACKET_TRANSFER !M1 !M0 !READ !A0 !REQ_PACKET_TYPE !NIL_DATA !NETRESP_NIL !OUTQI0 !M1 !OUTQI0",3)
(3,"LMD_PUT !M0 !A0 !RCC_SH !FLAG (FALSE, TRUE), (PASS)",4)

IUT(3),TC(3): PASS
```

5.4 Results on using 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. The tester package brings solution to this problem.
All the test cases generated by TGV have been executed in the testing SIM1 environment. For each test case and the corresponding probe output file, the testing activity is almost instantaneous.

6 Conclusion

In this paper, we have presented an end to end industrial case-study concerning the automatic generation of executable tests suites for the Cache Coherency Protocol of the BULL's CC_NUMA Architecture. From the formal specification in Lotos language of this architecture and formalized test purposes, we have generated abstract test suites using the prototype TGV. The generated test cases have been experimented in the real testing environment of BULL's CC_NUMA architecture using the software toolset (called the tester package) we have developped. At this stage of the experiment, we have covered all the test purposes described in the test plan, except those requiring an interactive approach. In order to cover all the test plan, some improvements are needed for both TGV and the tester package, such as:

- the introduction of cycles in the test cases in order to reduce the inconclusive cases; this should improve the quality of the generated test cases,
- some tests need be executed in an interactive way; this requires to extend both the tester package and the testing environement.

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 industrial experiment also demonstrates that the prototype TGV which was developped for conformance testing of communication protocols can also be efficiently used to generate tests for hardware architectures.

Acknowledgements : This work has been done in the framework of DYADE, the Bull-Inria Research Joint Venture. It has been supported by the Bull R&D PowerPC(TM) technology Platforms Division, headed by Angelo Ramolini. Special thanks to Pierpaolo Maggi and Paolo Coerezza and all the methodology office in Pregnana for their help in providing the inputs for our case study. We also want to thank all the member of the TGV-TEAM of Irisa-Rennes/Pampa & Verimag-Grenoble/Spectre, and particularly Pierre Morel, Thierry Jéron & Claude Jard, for the time they spent in improving "on-the-fly" TGV during this experiment.

References

[1] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming - Special Issue on Industrial Relevant Applications of Formal Analysis Techniques, To appear*, 1997.

[2] L. Doldi, V. Encontre, J.-C. Fernandez, T. Jéron, S. Le Bricquir, N. Texier, and M. Phalippou. Assessment of automatic generation methods of conformance test suites in an industrial context. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *IFIP TC6 9th International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1996.

[3] M. Faci and L. Logrippo. Specifying Hardware in LOTOS. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of 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 the Joint International Conference on Formal Description Techniques for Distributed Systems and Protocols, and Protocol Specification, Testing, and Verification 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] T. Jéron and P. Morel. Abstraction et déterminisation à la volée : application à la génération de test. In G. Leduc, editor, *CFIP'97 : Colloque Francophone sur l'Ingénierie des Protocoles*, pages 255–270. Hermes, September 1997.

[7] E. Brinksma and T. Bolognesi. Introduction to the ISO Specification Language LOTOS. In *Computer Networks and ISDN Systems, Vol. 14*, pages 25–59, North-Holland, 1987. Elsevier Science Publishers B.V.

[8] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A Tool Box for the Verification of Lotos Programs. In *14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[9] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specification. In L. Logrippo, R. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification*, pages 379–394, Amsterdam, North-Holland, June 1990.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, and J. Hennessy. The Directory-Based Cache Coherency Protocol for the DASH Multiprocessor. Technical Report CSL-TR-89-403, Computer System Laboratory, Stanford University, CA 94305+, 1989.