System Design of a CC-NUMA Multiprocessor Architecture Using Formal Specification, Model-Checking, Co-Simulation, and Test Generation

Hubert Garavel, César Viho, Massimo Zendri

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation

Hubert Garavel — César Viho — Massimo Zendri

N° 4041

November 2000

THÈME 1

ISSN 0249-6399

System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation

Hubert Garavel* , César Viho† , Massimo Zendri‡

Thème 1 — Réseaux et systèmes
Projet Vasy

Rapport de recherche n° — November 2000 — 36 pages

Abstract: The application of formal methods to system-level design of hardware components is still an open issue for which concrete case-studies are needed. We present here an industrial experiment concerning the application of the process algebraic language LOTOS (ISO standard 8807) to the design of POLYKID, a CC-NUMA (*Cache Coherent – Non Uniform Memory Access*) multiprocessor architecture developed by BULL. The formal descriptions developed for POLYKID have served as a basis not only for model-checking verification using CADP (CÆSAR/ALDEBARAN DEVELOPMENT PACKAGE), but also for hardware-software co-simulation using the EXEC/CÆSAR tool, and for automatic generation of executable tests using the TGV tool.

Key-words: cache coherency, CC-NUMA, code generation, co-design, computer architecture, conformance testing, co-simulation, formal methods, formal specification, hardware design, LOTOS, process algebra, NUMA, rapid prototyping, system level design, test generation, testing, validation, verification.

A final version of this research report is to appear in the Springer-Verlag international journal *Software Tools for Technology Transfer.*

* INRIA Rhône-Alpes, 655 avenue de l'Europe, F-38330 Montbonnot St Martin, France, E-mail: hubert.garavel@inria.fr, Web: http://www.inrialpes.fr/vasy

† IRISA/IFSIC Université de Rennes I, Campus de Beaulieu, F-35042 Rennes cedex, France, E-mail: viho@irisa.fr, Web: http://www.irisa.fr/pampa

‡ BULL R&D, Via ai Laboratori Olivetti, I-20010 Pregnana, Milanese, Italy. Now at ST Microelectronics, 5 chemin de la Dhuy, F-38240 Meylan, France, E-mail: massimo.zendri@st.com

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Conception au niveau système d'une architecture multiprocesseurs CC-NUMA avec spécification formelle, vérification, co-simulation et génération de tests

Résumé : L'application des méthodes formelles à la conception au niveau système de composants matériels est un problème ouvert pour lesquel des études de cas concrètes sont nécessaires. Nous présentons ici une expérience industrielle concernant l'application de l'algèbre de processus LOTOS (norme ISO 8807) à la conception de POLYKID, une architecture multiprocesseurs CC-NUMA (*Cache Coherent – Non Uniform Memory Access*) développée par BULL. Les descriptions formelles développées pour POLYKID ont servi à la vérification basée sur les modèles en utilisant CADP (CÆSAR/ALDEBARAN DEVELOPMENT PACKAGE), à la co-simulation matériel-logiciel en utilisant l'outil EXEC/CÆSAR tool, ainsi qu'à la génération automatique de tests exécutables en utilisant l'outil TGV.

Mots-clés : algèbre de processus, architecture d'ordinateur, CC-NUMA, co-conception, co-simulation, cohérence de caches, conception de matériel, conception système, génération de code, génération de tests, LOTOS, méthodes formelles, NUMA, prototypage rapide, spécification formelle, test de conformité, validation, vérification.

1 Introduction

Hardware designers are confronted to important challenges due to the increasing complexity of hardware systems, in addition to the permanent constraints of reducing costs and time-to-market. These challenges can only be overcome by adopting higher level design approaches combined with computer-aided tools that improve automation, enable teams to manage complex designs, allow design blocks to be reused, and increase confidence in the correctness of the implementation.

There are various levels in hardware design. A distinction should be made between the uppermost design level (*system level*), which is mainly about the global design of an architecture and of the communication protocols between hardware entities, and the lower levels (*behavioural level, register transfer level, gate level*) used to produce an implementation.

Today, the industrial practice for dealing with the lower levels is strongly established and supported by commercial tools: the design of a circuit is typically described using a hardware description language such as VHDL [IEE93] or VERILOG [IEE95]; this description can be used for simulation, verification, automatic circuit synthesis, and testing.

On the opposite, the design of complex hardware systems at system level has not reached the same degree of maturity and, in many respects, is still an open problem. Although it is clear that higher-level formalisms with abstraction capabilities are needed to manage the increasing complexity of hardware systems, there is no general consensus on the appropriate formalisms to be used. A recent survey [LSVS00] on this issue reviews several formalisms (*models of computation*) that are potential candidates for system level design, such as discrete-event systems, dataflow process networks, Petri nets, synchronous/reactive languages, synchronous/hierarchical FSMs (StateCharts), process algebras, distributed abstract state machines, timed/hybrid automata, etc. Although certain candidates can easily be dropped from the list because they do not match usage requirements or do not scale up to large designs, the number of experiments is probably not large enough yet to have a clear vision of the most suitable formalisms for system level design.

In this report, we present an industrial case study illustrating the application of formal methods to the system-level design, validation, and testing of a multiprocessor architecture named POLYKID developed by BULL.

This case study was tackled in the context of the VASY (*Validation of Systems*) project of DYADE, the BULL-INRIA Joint Venture for Advanced Research. DYADE focuses on technology transfer: INRIA researchers and BULL engineers work together in common projects in which the scientific results of INRIA are applied to problems of interest to BULL. Each project within DYADE must be funded by (at least) one business unit of BULL; this requirement is meant to ensure the industrial relevance of the proposed work.

The three main scientific goals of this case study are:

- to assess the suitability of process algebras — specifically, the LOTOS language [ISO88, BB88] — for the design at the system level of industrial-size hardware architectures and protocols;

- to experiment the toolbox CADP [FGK+96] (developed by INRIA Rhône-Alpes and the VERIMAG laboratory) for model-checking verification of hardware protocols;
- to extend the methods and algorithms of the test generation tool TGV [FJJV96, FJJ+97] (developed by INRIA Rennes and the VERIMAG laboratory) originally designed for telecommunication protocols described in SDL, in order to use it also for hardware systems specified in LOTOS.

The BULL-INRIA co-operation on formal methods for hardware systems started in 1995 with a feasibility study targeted at assessing the applicability of LOTOS and CADP to the formal specification and verification of the POWERSCALE bus arbitration protocol [CGM+96]. The feasibility study convinced BULL engineers that model-checking verification could detect design errors at an early stage in a product life cycle, and thus could be introduced in the design of hardware protocols profitably.

Based on these positive results, it was decided to continue the collaboration on a different, larger case study, the POLYKID architecture. The work on POLYKID described in the present report differs from the previous feasibility study in several respects:

- The first motivation was to apply formal methods to an architecture under development rather than to a frozen design already embedded in a commercial product (as it was the case for the POWERSCALE protocol). This would imply stronger interactions with BULL developers, as well as greater constraints to follow the technical and marketing changes brought to the project.
- The second motivation was to go beyond mere model-checking verification in order to encompass a broader part of the design life cycle. A more ambitious approach based on formal methods was sought, which we can summarize as follows:
  - At the system level design, the correctness of the architecture and associated protocols was to be formally described using LOTOS and verified using the model-checking tools of CADP;
  - At the lower design levels, the correctness of the VHDL implementation was to be checked using the traditional methods used by BULL and the hardware design community, including simulation and manual testing;
  - But the goal was also to establish a connection between those different design levels, namely by reusing the LOTOS descriptions developed at system level for: (1) producing software emulations of hardware components in order to perform *hardware/software co-simulation*, (2) generating tests automatically in order to check the VHDL implementation or even the real circuit. Co-simulation and automatic test generation are new means to create value from the introduction of formal methods in the development.

This report is organized as follows. Section 2 gives an overview of the POLYKID architecture and of its cache coherency protocol. Section 3 deals with the formal specification and

verification: it also justifies the choice of LOTOS as a specification language for system level design and presents the CADP tools used in the POLYKID experiment, as well as related work on the verification of cache coherency protocols. Section 4 describes the technical approach followed to produce a software emulation of the remote cache controller of POLYKID and its use for hardware/software co-simulation. Section 5 presents the principles of the TGV tool, the application of this tool for automatic test generation, and the execution of the generated tests in the real POLYKID testbench. Finally, Section 6 summarizes the main conclusions of the POLYKID study and suggests directions for future work.

2 Description of the CC-NUMA architecture and its cache coherency protocol

2.1 CC-NUMA architectures and cache coherency protocols

Multiprocessor computers aim at obtaining high performances in a scalable way. Depending on the way processors exchange data, there are basically two kinds of multiprocessor architectures: message-passing architectures, in which every processor has its own private memory and communicates with the other processors by sending and receiving messages, and shared memory architectures, in which all processors share a common address space.

A compelling feature of shared memory architectures is that existing sequential programs need not be rewritten when switching from an uniprocessor to a multiprocessor machine. One often distinguishes between two classes of shared memory multiprocessors: UMA (*Uniform Memory Access*) architectures — also known as SMP (*Symmetric MultiProcessors*), in which every process can access every memory location in a fixed amount of time, and NUMA (*Non Uniform Memory Access*), in which memory is organized hierarchically, so that some memory locations (e.g., processor local caches) can be accessed faster than others. NUMA architectures are of interest because they can potentially scale to a greater number of processors than UMA ones.

For efficiency reasons, a data stored in memory might be replicated in several copies stored in processor caches. The existence of multiple copies raises well-known issues (see, e.g., [HP96] for an overview of coherency and consistency issues). A NUMA architecture is said to be *cache coherent* (noted CC-NUMA) if it maintains coherency between the various copies of the same data. Depending on the topology of the architecture, there are two main classes of cache coherency protocols:

- *Bus snooping protocols* work for multiprocessor systems organized around a bus. In this approach, each processor continuously scrutinizes the bus to be aware of *bus transactions* (e.g., read and write commands) issued by other processors. Most commercial systems rely on the MESI cache coherency protocol (see, e.g., [PH97, chapter 9]), which is used in INTEL's PENTIUM PRO and IBM's POWERPC microprocessors notably.

  The MESI protocol implements a refined form of mutual readers-writers exclusion algorithm: multiple processors are allowed to store in their caches local copies of memory

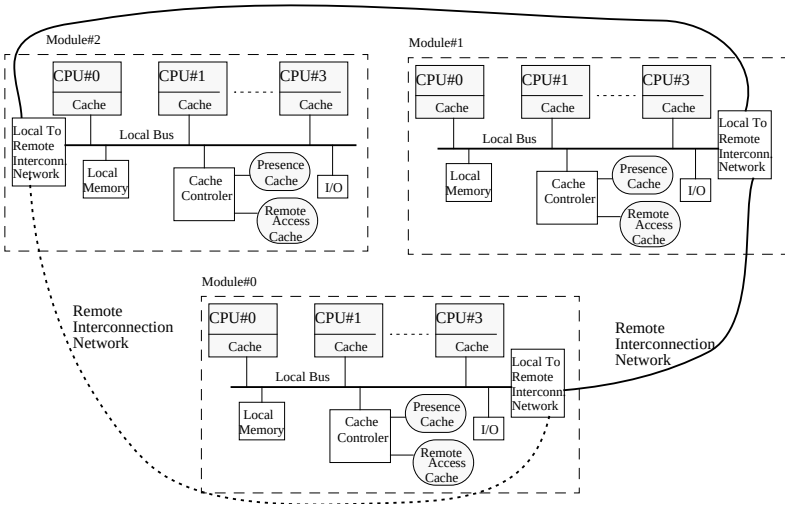

Figure 1: The POLYKID Architecture

objects, but only a single one should modify the object at a time; when reading an object, processors must make sure that they access the most recent copy; additionally, to reduce demand on bus bandwidth, processors must avoid memory accesses as much as possible by exchanging and updating the contents of their respective caches. The name MESI stands for the names of the four possible states of cache contents: *Modified, Exclusive, Shared*, and *Invalid*.

- *Directory-based protocols* can be used for networks of processors not connected to a bus, a situation that prevents processors from snooping to acquire a global knowledge of the transactions done. Instead, the cache coherency protocols maintain data structures named *directories* that contain status information related to main memory objects for which copies exist in the processor caches. For instance, directories can be used to retrieve the set of processors that have a local copy of a given object. In scalable, cache coherent architectures, directories are distributed with the processors and memories.

2.2 Overview of the POLYKID architecture

POLYKID was an experimental CC-NUMA multiprocessor architecture developed by the Unix Servers division of BULL in Pregnana (Italy). A working prototype of a POLYKID machine running the AIX operating system was built in 1998 and found to perform correctly. Although this machine was technically operational, it was not made commercially available due to marketing and timing considerations.

A POLYKID machine consists of a scalable collection of *modules* (typically 4, 8, 16, etc.) interconnected by a double ring network implementing the (lower layers of) the SCI (*Scalable Coherent Interface*) [IEE92] standard, which is ubiquitous in high-end servers.

Each module of POLYKID is built simply by reusing an existing BULL multiprocessor machine already available (named PEGASUS), in a slightly modified version with a reduced number of processors. A POLYKID module consists of 4 *nodes* connected by a data bus and an address bus. The data is implemented as a *crossbar*, i.e., a 4×4 matrix of switches allowing every processor to communicate directly with every other processor, thus providing a higher bandwidth than with a standard bus. Amongst the 4 nodes, two are dedicated to the processors, one is dedicated to the main memory and one is dedicated to PCI bus management and input/output devices. Each processor node contains two POWERPC microprocessors (model 620) connected by the standard POWERPC system bus. Each POLYKID module is thus a complete SMP machine supporting 4 processors (instead of 8 in a PEGASUS machine). A simplified schema of the POLYKID architecture (in which nodes and crossbars are not represented explicitly) is given on Figure 1.

As regards cache coherency, each module implements a MESI protocol internally. To achieve cache coherency between different modules (as the main memory is distributed among the modules), the (optional) cache coherency features existing in SCI might have been used; however, these features — specified as fragments of C code in the SCI standard — were found to be difficult to implement in hardware.

Therefore, in order to maximize performance, the POLYKID architects decided to retain from SCI only the physical and logical layers (for which commercial chips were available), but not the cache coherency layer. Instead, the POLYKID architects designed their own distributed, directory-based cache coherency protocol inspired from Stanford's DASH architecture [LLGH89], the forerunner of CC-NUMA computers.

This cache coherency protocol is a key feature of the POLYKID architecture. Each module is equipped with a *presence cache*, i.e., a cached directory that maps all the memory blocks cached outside the module, and a *remote cache* which, in order to improve performance, stores locally the most recently used blocks retrieved from remote memories as well as their current status (invalid, shared, or modified). The cache coherency protocol of POLYKID is directly implemented in hardware: BULL designed for this purpose a special ASIC (*Application-Specific Integrated Circuit*) named RCC (*Remote Cache Controller*) that is in charge of cache coherency. Each module has one RCC circuit, physically located on the interconnection board that establishes a bridge between the POWERPC system bus and the SCI network.

Cache coherency protocols are inherently complex, and the POLYKID protocol is no exception. Complexity comes from the asynchronous concurrency between the processors, the hierarchical structure of the architecture (there are several modules, each module containing nodes containing themselves processors), the existence of several levels of cache, and the many different situations that must be considered (especially, to handle collisions).

Consequently, the complexity of the RCC described in VHDL register transfer level was important (300,000 gates approximately). Because of this complexity, and because the

correctness of the cache coherency protocol was essential to a proper functioning of the architecture, the designers of POLYKID agreed that this protocol was the right target for introducing formal methods in the design of their architecture.

3 Formal specification and verification

3.1 Introduction

As the reference specification of the POLYKID architecture was only provided under textual form, it was necessary to develop a formal description of the cache coherency protocol on the basis of this reference informal specification. There were three main motivations for producing such a formal description:

- to improve the understanding of the architecture and the quality of its reference documentation,
- to verify critical correctness properties (e.g., deadlock freeness, cache coherency, etc.) using model-checking verification tools that can not work with informal specifications,
- to obtain an error-free formal description suitable for the test generation tool TGV, as this tool assumes that the formal description given as an input is a correct model of the system to be tested.

It was then decided to use the language LOTOS for the formal specification of the POLYKID cache coherency protocol. In the next section, we briefly present LOTOS and justify its choice for the POLYKID project.

3.2 The Formal Description Technique LOTOS

LOTOS is a Formal Description Technique for specifying communication protocols and distributed systems at a high abstraction level and with a strong mathematical basis. Developed during 1981–1988 (especially in the framework of the European ESPRIT project SEDOS), its definition was standardized by ISO (*International Organization for Standardization*) in 1988 [ISO88]. Several tutorials for LOTOS are available, e.g. [BB88, Tur93]. LOTOS features two clearly separated parts:

The data part, intended to the description of data structures, is based on the theory of algebraic abstract data types, namely on the ACT-ONE specification language [EM85, dMRV92]. Data structures are described by *sorts*, which represent value domains, and *operations*, which are mathematical functions defined on these domains. The meaning of operations is defined by algebraic *equations*. Sorts, operations, and equations are grouped in modules called *types*, which can be combined together using importation, renaming, parametrization, and actualization. The underlying semantics is that of initial algebras [EM85].

The behaviour part, intended to the description of concurrent processes that synchronize and communicate by message-passing rendezvous, is based on the process algebra approach for concurrency, and combines the best features of the process calculi CCS [Mil80, Mil89] and CSP [Hoa85]. LOTOS has a small set of basic operators representing the primitive concepts of concurrent systems: sequential composition, non-deterministic choice, guard, parallel composition, etc. The language is fully compositional, as complex behaviours can be obtained by combining elementary ones using these operators. As in most process algebras, the semantics of LOTOS is formally defined in terms of *labelled transition systems* [Par81] (or simply *graphs*), i.e., directed graphs whose vertices denote the global *states* of the system and whose edges correspond to the *transitions* permitted by the system.

At the very beginning of the POLYKID project, the choice of LOTOS was motivated (in part) by the wish to establish a collaboration between BULL engineers seeking for the benefits of formal methods and INRIA researchers developing compilers and verification tools for LOTOS. From the experience gained during the POLYKID project, we are still convinced that this choice was correct. When considering the possible models of computation for system level design (a survey can be found in [LSVS00]), many of them are not optimal for cache coherency protocols:

- Traditional hardware description languages (such as VHDL or VERILOG) are not appropriate, even if they provide various abstraction levels, e.g., the behavioural style of VHDL. Because these languages are more oriented towards implementation rather than specification, they are much too detailed for cache coherency protocols: in the case of POLYKID, the most primitive actions of the protocol are bus transactions and network packet transfers, which should be described at a higher granularity level than VHDL, in which every signal change has to be specified at every clock cycle. As pointed out in [CSB00], higher level formalisms with abstraction capabilities are required in order to avoid such over-specification issues.

  Also, the complexity of POLYKID was beyond the capabilities of commercial tools available for verifying VHDL and VERILOG designs: as mentioned in Section 2, the estimated complexity of a single RCC circuit is 300,000 gates and there are as many RCCs as modules in the system. Additionally, the discrete-event model used in those tools contains more information (e.g., timing information) than necessary for the verification of cache coherency.

- Languages with a synchronous semantics [Hal93] (such as ESTEREL, LUSTRE, SIGNAL, STATECHARTS, COSPAN's S/R model, etc.) are not appropriate for cache coherency protocols, even if they can be useful to design other kinds of hardware systems. The reason is that the global behaviour of a cache coherency protocol is concurrent, asynchronous, and non-deterministic: every processor in every module decides to read and write memory objects at its own rate, depending on the application program being executed on this processor. Memory access requests are emitted in any order and without correlation.

- Languages based on communicating finite state machines connected by infinite FIFO queues (such as SDL or the CFSM formalism of [LSVS00]) are not suitable. There are indeed some FIFO queues in the POLYKID architecture, but only of bounded size (for instance, the POWERPC output queue may contain at most 8 bus transactions) and not on every communication link. For those queues, blocking send and receive primitives are required: a processor trying to get a transaction from an empty queue or to put a transaction in a full queue must wait until the queue contents evolve, or must do something else.

On the opposite, there are strong scientific reasons to use a process algebra such as LOTOS for the system level design of cache coherency protocols:

- Process algebras enable protocol description at a higher level than traditional hardware description languages. Their underlying semantic model, i.e., labelled transition systems, is simpler and more abstract than event-discrete systems. It is worth noticing that other formalisms used for system level design (e.g., grammar-based methods, abstract state machines, etc. [CSB00]) present important similarities with process algebras, but the theoretical models and semantics of concurrency are perhaps more elaborate in process algebras.

- The synchronization mechanism of LOTOS, which combines rendezvous synchronization and message-passing communication in the line of Hoare's CSP, is well suited for the specification of hardware entities such as processors, memory controllers, bus arbiters, etc. (see [CGM+96] for an example). The electrical signals used to establish communications between these components are easily described as rendezvous interactions between LOTOS processes. Moreover, the multiway rendezvous mechanism of LOTOS is general enough to express different communication mechanisms (such as (infinite or bounded) FIFO queues, prioritized access to a shared resource like a PCI or SCSI bus, etc.) as derived cases.

- Cache coherency protocols often use more complex data structures than mere booleans, integers, and enumerated types. For instance:

  - the parameters of bus transactions are discriminated union types, the fields of which vary in number and types;
  - directory structures are arrays (or associative arrays) mapping memory blocks to cache status information;
  - FIFO queues are arrays or lists of elements, themselves of structured types.

  Such complex data structures can be expressed using the algebraic abstract data types of LOTOS. Moreover, the CÆSAR model-checker for LOTOS included in the CADP toolbox is one of the very few model-checkers capable of handling dynamic data structures (such as lists, trees, etc.) that rely on run-time memory allocation.

Besides these motivations closely related to cache coherency protocols, there are additional reasons for choosing LOTOS, especially the fact that LOTOS is a stable, international standard, that process algebras have strong theoretical foundations enabling different degrees of verification (from model-checking to theorem proving) and using different approaches (bisimulation relations, temporal logics, refinement and compositional verification, etc.).

For completeness, we can mention different approaches [FL93, ST93, HT00, YG98] in which LOTOS is used to specify hardware components at a much lower level than system level (e.g., at gate level). Although these approaches compete directly with established hardware description languages and associated verification methods, they can be useful for the design and verification of so-called *asynchronous* circuits.

In the next section, we present the CADP tools used to compile, execute, and verify the LOTOS descriptions developed for the POLYKID architecture.

3.3 Verification tools and methodology

The existence of computer tools capable of checking formal specifications automatically and efficiently is a positive factor for the dissemination of formal methods, especially in industrial projects: this is the motivation behind the design of the CADP tools used during the POLYKID project. These tools allow incremental degrees of correctness checking, with different limitations depending on the complexity of algorithms and the cost of data structures involved:

- The first degree of checking is provided by the CÆSAR.ADT [Gar89] and CÆSAR [GS90] compilers. These two LOTOS compilers share a common front-end part, which performs syntactic and static semantics analysis of the LOTOS specifications under study. Both compilers are complementary and differ by their back-end parts: CÆSAR.ADT handles the data part of LOTOS descriptions, whereas CÆSAR handles the behaviour part.
- The second degree of checking is obtained by translating the LOTOS description into executable C code using the CÆSAR.ADT and CÆSAR compilers. The generated C code can be used for several purposes (simulation, random execution, *on the fly* verification, test generation, etc.) as it complies to the principles and application programming interfaces of the OPEN/CÆSAR software architecture [Gar98]. Amongst the many OPEN/CÆSAR application tools included in the CADP toolbox, two have been used intensively during the POLYKID project:
  - XSIMULATOR is a graphical, interactive simulator that allows step by step execution of a LOTOS description. This tool was used to find mistakes at the specification level.
  - EXHIBITOR is a verification tool that searches on the fly for execution sequences matching a regular expression pattern. This tool was used to find erroneous sequences leading to a coherency paradox in the cache protocol.

  In practice, as regards the limitations:

- There is almost no limitation with CÆSAR.ADT: all LOTOS descriptions, whatever their size, compile without problem, and the quality of the C code generated for LOTOS sorts and operations remains satisfactory in size and performance.
- In most cases, there is no limitation with CÆSAR for C code generation, except for some LOTOS descriptions containing "too many" parallel processes and potential synchronizations between these processes, a situation that occurred only when modelling the entire POLYKID architecture (see Section 3.4 below). This limitation is inherent to the compiling algorithms of CÆSAR, based on the translation of LOTOS specifications into Petri nets extended with variables, conditions, and actions [GS90]: when the number of process synchronizations increases, the number of Petri net transitions increases as well.
- There is no limitation with XSIMULATOR, as this tool only stores in memory the states visited since the initial state. On the opposite, the memory requirements of EXHIBITOR strongly depend on the complexity of the state space and the regular expression pattern to be searched: therefore, EXHIBITOR may run out of memory if the state space is too large and if the regular expression does not constraint the search enough.

- The third degree of checking is based on exhaustive model-checking verification. In addition to producing executable C code for a LOTOS description, CÆSAR can also generate the labelled transition system (or graph) corresponding to this description. Due to the well-known *state explosion* problem, there are limitations on the sizes of graphs that can be generated exhaustively. When this is possible, the graph obtained can be verified using various model-checking techniques, notably bisimulation relations and/or temporal logics.

  For the verification of POLYKID, we have mostly used the bisimulation approach supported by the ALDEBARAN [FM91] tool. ALDEBARAN allows to minimize a graph modulo a bisimulation relation (e.g., strong bisimulation, observational equivalence, etc.) or to compare two graphs modulo a bisimulation relation. The former functionality is used to reduce the graphs produced by CÆSAR to a smaller, yet equivalent form; the latter is used to compare the graph modelling the system under study against various graphs modelling the various properties to be verified.

  However, for a complex system such as POLYKID, the state explosion problem is likely to occur, thus preventing graphs from being generated exhaustively. Fortunately, the compositional verification techniques proposed in [KM97] provide an effective solution to this problem.

  In this approach, the LOTOS specification to be verified is split into a set of communicating processes, composed together using LOTOS parallel operators and/or interface constraints. The splitting of the specification and the definition of interface constraints must be done manually. There are often several possibilities for splitting and interface definition; the choice between them is based on heuristics and requires insight

about the architecture of the system and the behaviour of its components. Then, each process is translated into a graph separately, taking into account the interface constraints, which avoid generating parts of the state space that are not needed. Then, ALDEBARAN is used to minimize each graph modulo a chosen bisimulation. Finally, the minimized graphs are combined altogether in order to produce the graph corresponding to the whole system. This compositional generation approach is implemented in the EXP.OPEN and PROJECTOR tools, which are part of the CADP toolbox.

3.4 Main results

The formal specification and verification activities of the POLYKID architecture in general, and the cache coherency protocol in particular, took about 9.5 man×months between February 1996 and September 1997. During this period, the architecture and the protocol were under design: the formal descriptions in LOTOS produced for POLYKID had to evolve regularly, in order to keep track of the changes introduced by the architects of POLYKID in their textual reference specifications. Taking this constraint into account, the specification and verification activities can be divided in three successive phases [Che97]:

- During the first phase (6 man×months), the protocol was under construction: many features were not frozen, including the most delicate points (e.g., collision rules).

  Two formal descriptions in LOTOS specifications were produced. The first one (4,000 lines) dealt with the whole POLYKID architecture. It was both extensive (describing most of the components of the architecture) and very detailed (although incomplete). It modelled a system with 4 modules, 2 processors per module with their local caches, the remote cache level and 12 different memory addresses. Unfortunately, this description was too large (66 processes, 52 gates, and 12 operations) for being compiled by the CÆSAR compiler.

  The second one (2,000 lines) focused on the cache coherency rules (without the collision rules, not available at that time). It modelled a system with 3 modules, 2 processors per module with their local caches, the remote cache level, 1 memory address, and the input and output queues merged. It was possible to compile this description (1 process, 11 gates, and 117 operations) using the CÆSAR compiler and to execute it step by step using XSIMULATOR, but not to generate the corresponding graph. Moreover, the description was still too detailed and not decomposable, so that compositional verification was not applicable.

  The main benefit of this first phase was to clarify undocumented concerns and to point out potential issues (in this phase, it is difficult to speak about errors, as the reference specification was neither stable, nor complete). About 55 questions were asked to POLYKID architects (40 orally and 15 in writing).

- During the second phase (1.5 man×months), the cache coherency protocol was complete in draft versions, so that its formal description in LOTOS was possible. About

20 questions were asked to the POLYKID architects to understand the last changes in the protocol and to get additional information.

The formal description activity and the use of the XSIMULATOR and EXHIBITOR tools enabled the detection of various several mistakes (e.g., typing errors and obvious omissions), which could easily be solved by a reader familiar with the protocol, and about which we do not report here. It also revealed 7 more serious issues, among which:

- 2 uncovered cases (i.e., some situations not handled by protocol rules, leading to unclear consequences),
- 4 undocumented features (i.e., protocol rules that were implicitly assumed to be implemented in the VHDL code, but not documented, and that are needed to guarantee a correct behaviour,
- 1 deadlock (typically, a read command that does not get the requested data).

- During the third phase (2 man×months), the cache coherency protocol was mature and fairly stable. In order to obtain a LOTOS description suitable for compositional verification, a new, thorough analysis of the reference specification was undertaken, raising about 10 questions to the POLYKID architects. This work revealed 13 serious issues, categorized as follows:

  - 6 undocumented features,
  - 1 deadlock,
  - 6 data consistency violations (two different and valid copies of the same memory location).

  The POLYKID architects modified the cache coherency protocol to fix these problems. Then, two new LOTOS descriptions (2,000 lines each) were built for the corrected protocol. Noticing that 5 out of the 7 errors above (deadlock and data consistency violations) would appear in a 2-module configuration without processor caches, these LOTOS descriptions were abstracted as much as possible to keep exhaustive verification tractable. They modelled a POLYKID system with 2 modules, 1 processor per module without local cache, 2 distinct memory addresses (colliding in the module cache) and distinct input and output queues described as 1-slot buffers. The two LOTOS descriptions differed by the number of bus operations permitted: 4 and 9, respectively.

  These descriptions were small enough (10 processes, 11 gates, 39 operations) and written in a decomposable way, so that compositional generation was tractable. The graph corresponding to 4 bus operations had 59,379 states and 216,539 transitions, and the graph corresponding to 9 bus operations had 230,561 states and 1,054,793 transitions. These graphs were obtained in 45 and 90 minutes, respectively, on a SUN ULTRASPARC-1 workstation with a 144 Mhz processor and 256 Mbytes of RAM. On each graph, it was verified that the following correctness properties were satisfied:

  - no deadlock,

– no coherency paradox due to collision rules,

– no coherency paradox due to memory aliasing.

In order to be sure that the model-checking approach was relevant, 2 of the 7 errors detected during the formal modelling phase were introduced in the LOTOS descriptions: the verification tools detected these errors and produced appropriate diagnostic sequences.

3.5 Conclusion

From the beginning, the formal specification and verification activities were considered as being part of the design and development of the POLYKID architecture. Although this is a good example of the growing acceptance of formal methods in the industry, it should be clear that maintaining the consistency between a textual reference specification and the corresponding formal description has a cost, and requires a tight interaction between the design team and the verification team, especially if both teams are in different locations. Ideally, this cost could be lowered if protocol designers would adopt formal methods themselves.

The use of a formal description technique such as LOTOS for specifying cache coherency protocols proved to be a valuable help for the design and debugging of the POLYKID multiprocessor architecture: 20 serious problems were found, including 8 behavioural issues such as deadlocks and consistency violations. Additionally, it helped to improve the quality of the project documentation, by clarifying ambiguities and identifying implicit assumptions.

It is worth noticing that these errors were detected very early in the design cycle. Because of their low probability of occurrence, some of these errors would certainly have been difficult to detect, reproduce, and understand using traditional techniques based on simulation and testing of the VHDL implementation.

It should be stressed that this experiment is in no way a formal proof of total correctness, because:

- it is performed on simplified, downsized configurations,
- the formal specifications could be erroneous,
- the list of properties to be verified could be erroneous or incomplete,
- the verification tools themselves could be erroneous.

In spite of these restrictions, it was agreed that the use of formal methods clearly increased the quality of the design, by detecting errors and by showing that, after correction, these errors would disappear from the modified versions of the cache coherency protocol.

4 Co-simulation using software emulation of hardware components

4.1 Introduction

Another important aspect of the POLYKID experiment deals with software emulation of hardware components.

When designing a large, complex multiprocessor architecture, it is desirable to have a complete working prototype as soon as possible in order to check the proper functioning of hardware and firmware, measure their performances, and enable operating system adaptation and tuning. This prototype can be used to run various tests, a significant test being the ability to boot, first the firmware, then the entire operating system successfully.

However, many different ASICs are required for a CC-NUMA architecture. Due to unexpected delays in chip design and/or production, some of the circuits may not be ready on time, thus delaying the availability of the prototype.

To overcome this problem, one can use the classical "hardware emulation" approach, which consists in replacing an unavailable ASIC by a dedicated machine that emulates the netlist of the missing ASIC. However, hardware emulation is not entirely satisfactory:

- it is very expensive, as the dedicated machine involves both specialized hardware, namely FPGAs (*Field Programmable Gate Arrays*) and dedicated software for mapping netlists to FPGAs;
- it is complex to configure, as netlists often do not fit on a single FPGA, thus have to be split on several FPGAs, which is a tedious process;
- it is not yet powerful enough to tackle large circuits with several hundred thousands of gates, so that ASICs must be simplified in order to be emulated.

Our experiment took place during the build process for the first prototype of POLYKID. At this time, an essential ASIC circuit, the RCC (see Section 2) was not available. The goal was to produce a software program that would emulate the behaviour of an RCC circuit.

The intended execution environment for the software emulation was the following. In each module, one POWERPC microprocessor was put in charge of executing the RCC emulator. This microprocessor did not run a full-fledged operating system, but only a simple monitor. The code of the RCC emulator was kept in a Flash EPROM and loaded into main memory upon initialization of the system. The data structures of the emulator were mapped into main memory. The other POWERPC microprocessors of the module kept their normal role and were used to run test programs. The RCC emulator communicated with its environment — the system bus and SCI network — using interrupts sent by a TSP (*Transponder*) circuit implementing the protocols for the system bus and SCI network. The role of this circuit was to route bus transactions and network packets from/to the RCC. The TSP was synthetized from VHDL code and implemented on an FPGA.

To take advantage of the existing LOTOS specification of the cache coherency protocol developed for verification purpose, it was decided to use LOTOS for modelling the RCC and to rely on the CÆSAR and CÆSAR.ADT compilers for generating C code automatically.

4.2 Main results

Several successive implementations of the RCC emulator have been produced. The first implementation consisted in 3,400 lines of LOTOS code and 7,000 lines of C code (both numbers including comments). The LOTOS code contained most of the logic of the cache coherency protocol, the data type part defining the states and transitions of the cache, and their relation with bus transactions, and the behaviour part encapsulating the cache coherency protocol into one sequential, deterministic process.

The hand-written C code implemented the interface between the RCC emulator and the TSP: basically, it defined low-level data structures (such as bit fields, network packets, bus signals, and input/output buffers to store network packets and bus signals), as well as routines to access and modify these data structures. It also provided some ANSI C library functions (such as `malloc` or `printf`) not available in the monitor environment.

To have the emulation work practically, it was necessary to establish a connection between the RCC emulator and its hardware environment, i.e., to establish a connection between the LOTOS description and the hand-written C code. This is an instance of a more general problem: how to interface a formal description written in a process algebra such as LOTOS with its external environment? This problem is of practical importance, but there are only a few related publications, even if process algebras have been often used for the description of real-life systems or devices.

To the best of our knowledge, none of the approaches proposed in the literature for the implementation of LOTOS was found to be applicable to RCC emulation.

Among the implementations of LOTOS that address the problem of communication with the environment, many of them implement a LOTOS description as a collection of concurrent tasks controlled by a centralized run-time scheduler, which manages the communications between the tasks as well as the communications between tasks and the environment. These solutions were not applicable to RCC emulation, as the run-time scheduler relies on facilities (such as the UNIX lightweight processes [Dub89] or a reliable message transfer service [Sjö91]) that were not available for RCC emulation: as mentioned above, the RCC emulator was to run on a POWERPC with only a monitor, which is much more primitive than an operating system.

Another approach used in the TOPO compiler [MdM88] consists in extending the LOTOS description with fragments of C code (called *annotations*) attached to rendezvous interactions; an annotation is executed when the corresponding interaction occurs. This approach was not found to be adequate for several reasons: communication with the environment is obtained by side effects in annotations, which can compromise the correctness of the whole if the C code written by the user is not "compatible" with the LOTOS description; moreover, the communication with the environment is very limited, as the environment can neither refuse

an interaction permitted by the LOTOS description, nor select between several permitted interactions.

A third approach is implemented in the CÆSAR.ADT compiler, which allows to interface LOTOS code with hand-written C code. In the LOTOS description, certain sorts and/or operations can be declared as "external" using special syntactic annotations. The definition of those external sorts and operations is not given in LOTOS, but directly in C (as a collection of types, functions, and/or macro-definitions). CÆSAR.ADT generates no C code for external sorts and operations: it simply includes the C files that contain the hand-written implementation provided by the user. Although this mechanism is flexible and practically useful, it could not be used for RCC emulation, because the C functions implementing external LOTOS operations cannot perform side effects (as LOTOS operations themselves do not modify the state of the system), and because all the communications between the RCC and its environment (i.e., the system bus and SCI network) were modelled using the behaviour part of LOTOS.

Therefore, a novel scheme was designed, which provides a general interface mechanism between (the C code generated from) a LOTOS description and its environment. This scheme, called EXEC/CÆSAR, was fully implemented in the CÆSAR compiler and made available to the users of CADP since 1997. The principles of EXEC/CAESAR are the following:

- Communication between the LOTOS description and the environment takes places at the external (i.e., visible) *gates* of the LOTOS description, a gate being a point for rendezvous interaction.

- For each external gate G, the user must provide a corresponding C function with the same name `G`. In the case of the RCC, these C functions implement the TSP routines that access the system bus and the SCI network.

- Each function `G` has parameters corresponding to the data sent and received on gate G. In a first approximation, an output on gate G of an expression V (noted "G `!`V" in the LOTOS description) will translate to a C function call "`G (V)`" (where `V` is a *call by value* parameter) and an input on gate G of a value of sort S to be stored in variable X (noted "G `?`X:S" in the LOTOS description) will translate to a C function call "`G (&X)`" (where `X` is a *call by address* parameter to be modified in function `G`).

  In fact, the calling conventions are slightly more complex, as LOTOS allows to use the same gate with several inputs and/or outputs (e.g., "G `!`V_1 `?`$X : S$ `!`V_2"); moreover, the same gate G can be used several times with inputs/outputs that vary in number and types. This difficulty is solved by passing additional parameters to function `G`, which indicate the number, types, and mode (input or output) of parameters.

  Another complication is due to the fact that the formal semantics of LOTOS unifies inputs and outputs in order to define rendezvous between more than two processes. This problem was solved by modifying the CÆSAR compiler in order to preserve the distinction between inputs and outputs on the basis of the syntactic notations "`!`" and "`?`" used in the LOTOS description.

The EXEC/CÆSAR scheme does not handle LOTOS descriptions containing guarded inputs of the form "G ?X:S [P]", where P is a predicate defining the acceptable values for X. Guarded inputs express value negotiation between the LOTOS description and its environment, a feature we believe to be of little practical interest on external gates (at least, not needed for the RCC emulation).

- Each function `G` associated to an external gate G must return a boolean value, which is true if and only if the environment is ready to accept the rendezvous interaction on gate G with the proposed parameters.
- For a given LOTOS description, EXEC/CÆSAR generates a C program that behaves as a non-terminating polling loop. In a given state, the program determines the set of interactions permitted by the LOTOS description (i.e., the set of transitions going out of that state according to the formal semantics of LOTOS). If the set is empty, the program stops and signals a deadlock. Otherwise, it iterates on the elements of the set repeatedly until one is accepted by the environment; a round-robin mechanism is used to provide some fairness in the set enumeration order. For an interaction on an external gate G, the program queries the environment by calling the corresponding function `G` with appropriate parameters; the boolean result returned by the function determines whether the interaction is accepted or not. For an interaction on a internal (i.e., hidden) gate, the program does not query the environment: this kind of interaction is always accepted, as it models an internal branching decision rather than a rendezvous with the environment. As soon as one interaction permitted by the LOTOS description is accepted by the environment, the program performs the corresponding transition and moves to the next state.

The first version of the RCC emulator was ready in March 1997. When installed in the testbench environment at BULL Pregnana (Italy), the emulator was found to perform correctly, but slowly: it took about 100 ms to execute one cache transition (which consists of a sequence of 5–10 LOTOS interactions). This speed was clearly insufficient, as booting the AIX operating system on the testbench would have taken approximately one day, thus slowing down the whole test process.

Two main reasons for this problem were identified: first, the C code generated by the CÆSAR compiler was not fast enough (this code was originally designed for model-checking verification and targeted primarily at reducing memory footprint, not CPU usage); second, the POWERPC processor used in the testbench was running at a low frequence (15 MHz) due to technical/economical constraints on the POLYKID testbench. The fact that polling is inherently CPU-intensive was not a problem for RCC emulation, as one POWERPC processor was entirely dedicated to the execution of the RCC.

As the RCC emulation was on the critical path for the POLYKID project, it was decided to improve the behaviour part of the LOTOS description by introducing hand-written C code. First, only a single CPU-intensive process was rewritten in C. After this modification, the time needed to execute one cache transition dropped to 1.5 ms. Then, the whole behaviour part was rewritten in C and, after applying various optimizations based on the insight of

the system (such as tailoring the round-robin policy, reducing the buffer sizes, changing the communication mode between the RCC and the TSP), this time was further reduced to 0.5 ms. This final version of the RCC emulator consisted of 1,500 lines of LOTOS code and 8,000 lines of C code. The data part of the LOTOS description was kept unchanged, as the C code generated by the CÆSAR.ADT compiler was fast enough.

The emulator was used to perform various test suites. Beyond unit testing of the TSP hardware and the library routines, the whole system was tested first at the monitor level, then by booting the firmware, and finally the operating system. These experiences revealed several bugs in the RCC emulator code, in the TSP hardware, and in the communication mode between the RCC emulator and the TSP (for instance, interrupts were replaced by polling, as they created deadlocks). Most notably, electrical problems with the POWERPC 620 processor were discovered and 3 new defects in the POLYKID architecture were identified. Finally, after solving these problems, it was made possible to boot the AIX operating system on the testbench with 2 modules.

4.3 Conclusion

The development of the RCC emulator was recognized as an important technological step in the direction of hardware/software co-simulation: the BULL engineers involved in the POLYKID emulation project received a BULL *Eureka Research and Development Award* in 1997. The benefits of this experience are twofold:

- It demonstrates the feasibility and interest of combining software emulation and hardware, so as to obtain cheaper and faster prototypes than those using hardware emulation only.
- It enables the concurrent development of hardware, firmware, and operating system software: it makes possible to test the firmware and software very early in the design phase, even if some ASICs are not ready.

This co-simulation approach fits well within a LOTOS-based design methodology: parts of an existing LOTOS description developed for verification can be reused/adapted for software emulation using the EXEC/CÆSAR technology.

It was unfortunate that in 1997, when EXEC/CÆSAR was released, the C code generated from LOTOS was not fast enough, so that the behaviour part of the LOTOS description had to be rewritten in C manually. Today, there are good reasons to be optimistic about the feasibility of the approach:

- First, it should be noticed that the emulator was executed on a POWERPC processor running at a very low frequency (15 MHz, later raised to 45 MHz): a processor running at its normal clock speed would have given better results.
- In 1997, after a careful study of the C code generated by CÆSAR to understand where computation time was spent, several changes were brought to optimize the generated

C code (such as moving invariant computations out of loops). In the particular case if RCC emulation, these optimizations led to speed improvement by a factor ranging from 9 to 13. They also benefited to verification: combined with other improvements, the speed of model generation increased by a factor ranging from 2 to 160 (all these figures measured on an ULTRASPARC-1 workstation in Grenoble, not on the testbench in Pregnana).

- In 1999, further improvements were brought to the CÆSAR compiler, which reduced significantly the size of the extended Petri nets generated from LOTOS descriptions. Due to these improvements, the size of the Petri net corresponding to the RCC dropped from 137 places, 222 transitions, and 83 variables to 64 places, 149 transitions, and 68 variables, thus leading to the generation of more compact and efficient C code.

5 Automatic test generation and execution

5.1 Introduction

The last part of the experiment consisted in generating tests to be applied on the real implementation of POLYKID. The challenge in this step was to demonstrate that the TGV tool, originally developed for conformance testing of communication protocols, could also be used to generate tests for hardware architectures.

Following the methodology used at BULL for hardware testing, we started from an existing *test plan* document containing an informal description (in the form of tables with comments) of the main *test purposes* to be used for POLYKID (hereafter called the *system under test* and noted SUT for short). The test plan was designed with a good knowledge of the POLYKID architecture and attempted to check situations considered to be "at risk".

For instance, a test purpose describing an address collision situation is: "*The module M1 requests for a* FLUSH *transaction on the block address A0. The block address A0 is in module M0. Verify that the module M0 accepts the incoming* FLUSH *transaction. The CPU 0 of module M0 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 M0 and verify the good completion of the* FLUSH *transaction.*"

The test plan served as a basis for producing tests using two complementary approaches:

manual test generation: in this approach (also called *deterministic test generation*), tests are written by hand according to the test plan. The main limit of the approach is the productivity of test writers.

random test generation: this approach consists in sending random sequences of stimuli to the SUT. It can be improved by using software tools that guide random generation towards some particular situations listed in the test plan; this is done essentially by modifying the parameters of the stimuli. This approach has usually a low efficiency (measured in terms of bugs found per number of simulated cycles); however, it has the merit of creating unexpected high-traffic situations.

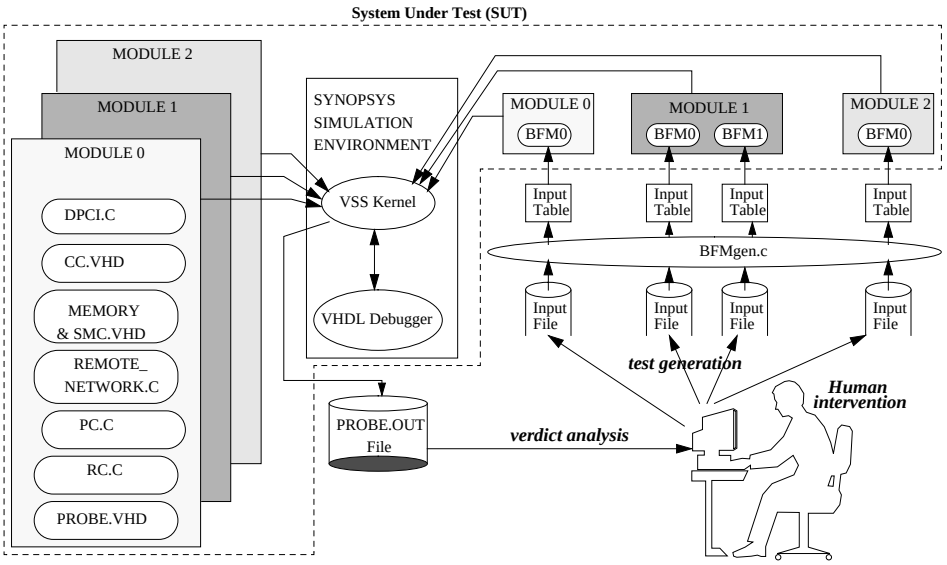

Figure 2: The SIM1 testing environment

There is another limitation in the standard methodology: human intervention is needed to check the outputs produced by the SUT in response to the commands sent by the tests (the expected, correct outputs being specified in the test plan). Such checking is tedious, expensive, and error-prone, as it is entirely based on informal specifications and on an informal notion of conformance between the test plan and the SUT. Moreover, the test plan can be erroneous itself, which questions the validity of the testing approach.

Automatic test generation addresses these various issues. In the next section, we present the testing environment used for POLYKID.

5.2 The testing environment

The testing environment of the POLYKID architecture (named SIM1) is described on Figure 2. It is composed of the VHDL description of the system under test (a POLYKID machine with 3 modules), the VSS (VHDL *Synopsys Simulator*) event-driven simulation kernel, and a front-end human interface (VHDL *Debugger*).

In each module, there is one *Bus Functional Model* (BFM) for each CPU. A BFM reads stimuli from its *input table*, which contains a sequence of transactions to be executed by the corresponding CPU. As the internal format of input tables is not directly readable by a human operator, an auxiliary program (named BFMGEN) is used to fill each input table from a corresponding *input file*, which is written manually, according to the informal test purposes specified in the test plan. Therefore, a test consists in a collection of input files, one per CPU. A practical difficulty when writing input files is the synchronization of the

CPUs. Synchronization is obtained by using *barrier transactions*, either for a single CPU or for all the CPUs, which prevents the CPU(s) from processing any further transaction before receiving responses for all the transactions already sent.

In response to input stimuli, the VSS writes output events to a file (named PROBE.OUT on Figure 2) at every clock cycle. The probe file contains for each module the sequence of actions executed, as well as the status of the presence cache and remote cache. Probe files are analyzed both visually (by comparing each line with the expected output informally specified in the test plan) and automatically (using software checkers developed specifically for the POLYKID architecture, which allow some kind of automatic verification of the behavior of the system).

From the test generation point of view, the whole testing environment can be seen as a black box that reads input files and generates probe output files. Most of the testing activity in the SIM1 environment takes place off-line (in batch) and consists of three successive steps: (a) stimulating the system, (b) collecting the reactions, (c) analyzing the reactions and emitting a verdict.

5.3 Principles of the TGV tool

TGV [FJJV96, FJJ+97, JM97] is a tool for automatic test generation based on conformance relations. To produce tests, TGV needs two main inputs:

- The first input of TGV is a reference executable description of the SUT, which can be given in various languages. Before the POLYKID experiment, TGV was mostly used for communication protocols described in SDL or given as finite-state machines. During the collaboration with BULL, TGV was modified to accept LOTOS descriptions, which was easy since TGV was, from the beginning, built on top of the language-independent OPEN/CÆSAR [Gar98] application programming interface. The input LOTOS description is compiled using CÆSAR.ADT and CÆSAR, which produce a C program containing the OPEN/CÆSAR primitives used by TGV for state space exploration.

  As the description given to TGV is assumed to be a reference model of the SUT, it should be strictly debugged and verified (for instance, as explained in Section 3) before test generation.

- The second input of TGV is a formal test purpose, represented as a finite automaton, the states of which can be either *normal*, *acceptance*, or *refusal* states, and the transitions of which are labelled by input or output interactions. A test purpose can be seen as an abstract view of the test case to be generated by TGV. Acceptance states tell TGV that the current test sequence is complete. Refusal states tell TGV to reduce state space exploration by cutting those transitions leading to a refusal state.

In practice, TGV often needs additional inputs:

- Depending on the properties to be tested, some interactions described in the LOTOS description can be found not important for the testing activity. These interactions can

be abstracted by using a *hiding file* that informs TGV that certain interactions are to be considered internal to the system and, thus, cannot be observed in the generated tests. Similarly, a *renaming file* passed to TGV modifies the interactions according to a set of substitutions based on regular expressions.

- Also, an *input/output file* given to TGV allows to distinguish between input interactions and output interactions. By default, the semantics of LOTOS makes no difference between inputs and outputs; however, the testing theory requires to make a clear distinction between *controllable events* (input stimuli from tester to SUT) and *observable events* (output reactions from SUT to tester).

The output of TGV is called a *test case*. A test case is a finite automaton, the states of which carry no special information, and the transitions of which are labelled by input or output interactions, and possibly by *test verdicts* (*pass*, *fail*, or *inconclusive*) which indicate whether the SUT conforms to its formal specification [FJJV96]. Each path of this graph describes a sequence of interactions (stimuli and reactions) between the tester and the SUT, and ends with a test verdict.

Generation of test cases requires various graph traversal operations (abstraction, minimization, determinization, and test case synthesis), which are performed on the fly (i.e., without generating the entire state space first) using the OPEN/CÆSAR technology. This is a key feature of TGV, which avoids, in many cases, the state explosion problem that makes other test generation tools not applicable to complex systems.

Contrary to the *batch* approach often used in hardware testing (in which the SUT receives all its stimuli first, the reaction being analyzed afterwards), the test cases generated by TGV are *reactive*, in the sense that the stimuli sent by the tester may depend on the reactions of the SUT observed in response to previous stimuli. Reactive testing increases the quality and coverage of tests, as more behaviors of the SUT can be tested.

5.4 Formalization of the system under test

To generate tests with TGV for the POLYKID architecture, a LOTOS description representing the SUT was needed. This description was obtained by reusing a LOTOS description written during the verification task (see Section 3.4) and adapting it to test generation by introducing abstractions described below. Turning the LOTOS description used for verification into a test-oriented description took about 1 man×month.

The resulting description (about 2,000 lines of LOTOS, one half for the data part, one half for the behaviour part) featured a POLYKID system with 3 modules, one POWERPC microprocessor per module, two distinct block addresses located in module M0 and two distinct data values. The main difference between this description and the original verification-oriented description was the introduction of abstractions, for two main reasons:

- The first reason is related to the size and complexity of the POLYKID architecture, which are likely to provoke state explosion, even though the OPEN/CÆSAR and TGV tools operate on the fly. Abstractions can be used as a way to reduce complexity by

hiding irrelevant details of the architecture. A typical abstraction in cache coherency protocols consists in merging sequences of events into a unique event. In a remote transfer, by example, a request from the sender is always followed by an indication for the receiver: both events can be merged.

- The second reason is that testing usually focuses on certain events only, so that other events can be hidden. For instance, a local response transaction always follows a local bus transaction although other events can take place between these two transactions. In the test-oriented LOTOS description, both transactions are modelled as a single event, and all other events are hidden.

These abstractions do not affect the test verdicts, since during the execution of the generated test cases, the same abstractions will also be applied to the probe output files produced by the SUT (see later the TRANSLATOR application in Section 5.7).

5.5 Formalization of the test purposes

Among the 7 groups of test purposes listed in the POLYKID test plan, we chose to focus on the groups 3 and 4 dealing with the cache coherency protocol implemented in the RCC circuit. It took 15 man×days to formalize all the test purposes in groups 3 and 4.

For instance, the test purpose given in Section 5.1 can be formalized as an automaton, a subset of which is displayed on Figure 3.

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 M1 requests for a FLUSH transaction on the block address A0. Input transitions are marked with a "?" symbol. State 5 is an acceptance state: when module M0 notifies the correct completion of the transaction by sending a response (noted NET_RESP_DONE) to module M1, state 5 is entered and TGV should consider that the test purpose is reached. Transitions labelled with a "*" label stand for "any other label". We do not represent the entire test purpose, which also contains one refusal state and many transitions leading from states 0..4 to this refusal state (a software tool was developed to produce these transitions automatically).

5.6 Automatic generation of test cases

For each test purpose listed in groups 3 and 4 of the POLYKID test plan, we have generated the corresponding test cases automatically using TGV. In total, 75 tests have been generated, most of which have more than 400 states and transitions. Because of this complexity, it is unlikely that such test cases could have been written by hand, even by experts.

Generating all these tests took nearly 1 man×month. The only problem faced during this task was the time taken by TGV to produce test cases, ranging from less than 1 second to 12 hours. This problem was due to the complexity of the POLYKID architecture; to speed up the test generation with TGV, some test purposes were refined to be less general and less abstract.

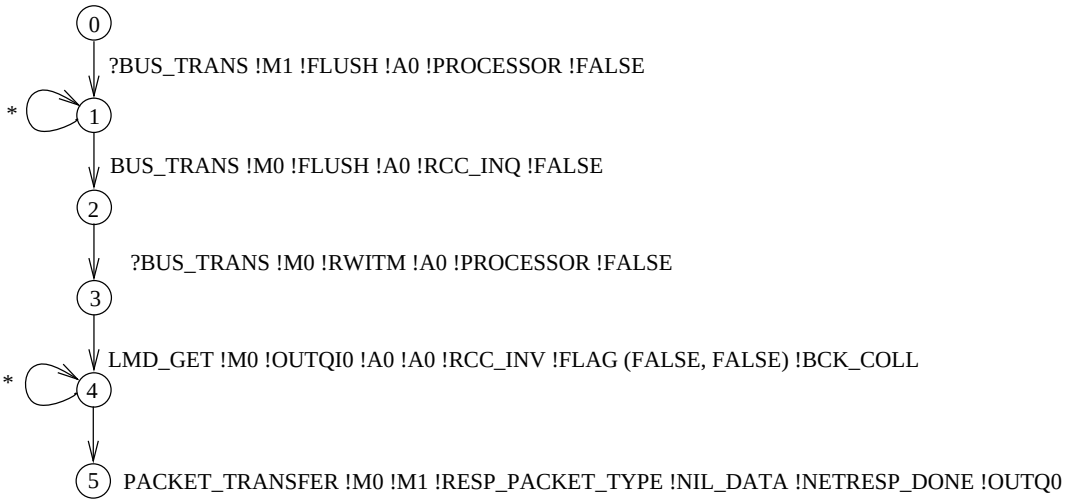

Figure 3: A simplified test purpose

The POLYKID experiment was the first in which TGV has been used on the fly. The practical interest of generating tests on the fly was clearly demonstrated, as it was impossible to produce a state graph for the whole POLYKID architecture.

During the experiment, the TGV tool was improved in several ways:

- Refusal states were introduced in the test purposes;
- Test cases were extended: previously, test cases could only be directed acyclic graphs, a constraint that appeared to be overly restrictive for testing certain functionalities of POLYKID. TGV was enhanced to allow cycles in test cases, which reduced the number of inconclusive verdicts and increased the test coverage.

This work was done by INRIA-Rennes in 8 man×months.

5.7 Batch execution of test cases

Although the test cases generated by TGV are reactive, the first goal was to implement a batch testing environment for the execution of these test cases. The challenge was to demonstrate to BULL engineers that the TGV-based approach could replace the usual approach based on hand-written tests. For this purpose, a *tester package* was developed, which is intented to automate the batch execution of tests in the SIM1 environment. The tester package is represented on Figure 4 and consists of three applications:

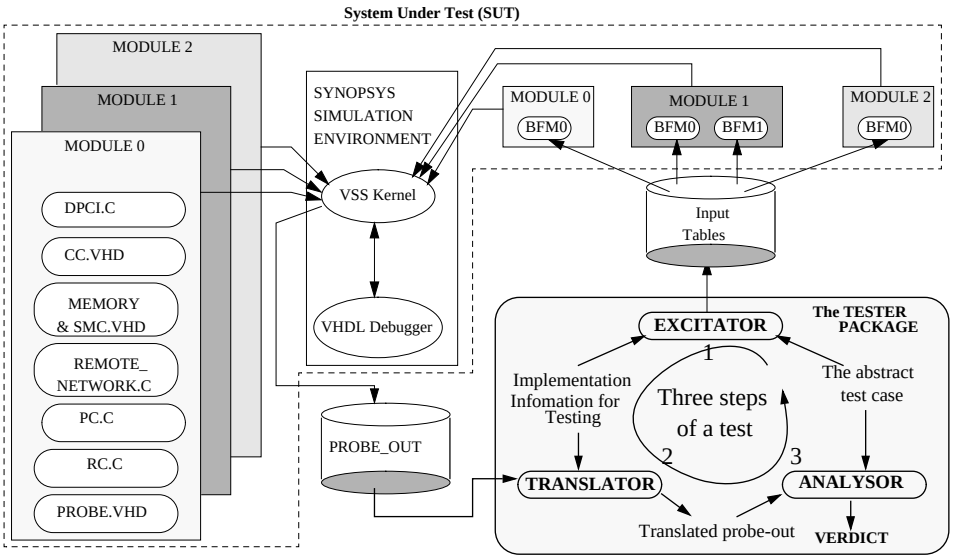

Figure 4: The POLYKID testing environment with the tester package

- EXCITATOR converts the stimuli contained in a test case generated by TGV (these stimuli are transition labels, encoded in the LOTOS syntax and making use of the identifiers defined in the LOTOS description of POLYKID) into the input table format readable by the BFMS. Once the conversion is done, EXCITATOR proceeds to the stimulation of the BFMS.
- TRANSLATOR converts the probe output file produced by the VSS kernel into a sequential execution trace, the transition labels of which are encoded in the LOTOS syntax. During this conversion, TRANSLATOR applies the same abstractions as those made on the LOTOS description.
- ANALYSOR compares the trace produced by TRANSLATOR against the test case generated by TGV, and emits a test verdict.

For each test case, the three applications of the tester package are launched manually and sequentially as indicated on Figure 4: first EXCITATOR, second TRANSLATOR, and third ANALYSOR.

Both EXCITATOR and TRANSLATOR take as input an IXIT (*Implementation eXtra Information for Testing*) that describes the mapping between the abstract data type values of the LOTOS description and the real data values of the SUT. This level of genericity permitted to reuse the tester package for another project without major effort. The development of the tester package by INRIA-Rennes took about 5 man×months; more details about the use of the tester package for batch testing of POLYKID can be found in [KVZ98].

5.8 Reactive execution of test cases

The batch execution approach developed in a first step was not entirely satisfactory, because the test cases generated by TGV are reactive. In particular, some tests could not be executed efficiently in batch mode. To keep the gain brought by the reactive nature of the tests generated by TGV, the tester package was enhanced to support reactive execution as well as batch execution.

The main difference introduced by reactive execution with respect to batch execution is that, for a given test case, the three applications of the tester package (EXCITATOR, TRANSLATOR, and ANALYSOR) are no longer launched only once, manually, and sequentially. In reactive execution, the applications are launched automatically and repeatedly until a stop condition is found:

- EXCITATOR is invoked on every stimulus found in the test case;
- TRANSLATOR is invoked at every clock cycle to translate output probe lines;
- ANALYSOR implements a synchronous automaton product between the test case and the SUT and performs checks at every clock cycle. For instance, if the SUT emits a reaction that does not exist in the test case, ANALYSOR report that the SUT does not conform to its formal description. When ANALYSOR reaches a test case transition labelled with a test verdict, the test execution stops and the remaining output probe lines are ignored.

The 75 test cases generated by TGV have been executed in the SIM1 testing environment using the tester package. The total time for executing all these tests was less than 20 hours (with approximately 1000 cycles per test, 0.6 second per cycle, and 5 minutes for loading and initializing the environment). For each test case, the overhead in simulation time due to the presence of the tester package was found to be negligible.

The reactive execution of the tests generated by TGV on the POLYKID implementation under test uncovered 5 bugs in the VHDL code (mostly about address collision and updates of the presence and remote caches). These bugs had not been caught by hardware testing experts using the traditional methodology. This revealed that test coverage was insufficient in the initial test plan, as some situations were not fully tested.

5.9 Conclusion

The work done during the POLYKID case-study established that TGV can be used to generate tests, not only for communication protocols described in SDL, but also for hardware architectures described in LOTOS.

Practically, the proposed approach for generating tests using TGV has one main benefit: it reduces the high cost of testing by increasing automation. Once the SUT and test purposes have been formalized, then test generation and test execution can be automated. The

time spent in describing the POLYKID architecture formally (mainly by reusing a LOTOS description previously developed for verification) and formalizing the test purposes was totally justified by the better quality of tests and the increased confidence in the implementation.

Technically, the proposed approach has the merit of introducing mathematical rigor in the testing process: all the entities used in testing (formal specification, test purposes, test cases, test verdicts, etc.), as well as the conformance relation between the SUT and its formal specification, have a well-defined meaning and semantics.

6 Conclusions

This report demonstrates the feasibility and benefits of using formal methods for hardware design at system level. More specifically, it illustrates how a language, LOTOS, originally intended to the formal description of communication protocols, together with tools developed for verifying and testing communication protocols, can enhance significantly the industrial approach to system-level design.

The report is based on a real-size application of formal methods in the development cycle of POLYKID, a prototype BULL multiprocessor CC-NUMA architecture. The experiment focused on the most complex part of POLYKID, the cache coherency protocol.

The LOTOS language was used throughout the whole case-study. Because of its message-passing semantics, LOTOS was found to be suitable for describing system-level aspects of multiprocessor architectures, such as bus transactions and network packet transfers. Its process algebraic foundations enabled the use of abstractions and bisimulation reductions to perform compositional verification. LOTOS was used first to produce formal descriptions of the cache coherency protocol (taking as a basis the reference documentation of POLYKID written in English). The LOTOS descriptions were used later for several purposes:

- model-checking verification of key correctness properties on reduced configurations of the protocol, in order to detect errors automatically and to increase confidence in the design,
- generation of embedded code to obtain a software emulation of a hardware component,
- generation of test cases to check the correctness of the VHDL implementation.

The proposed approach was effective in several respects:

- The formal specification and verification activities revealed 20 serious issues (among which 8 behavioural errors) in the cache coherency protocol for a limited cost (9.5 man×months on the industrial side); these issues have been admitted (and fixed) by the designers of POLYKID. Although some of these errors might have been found also by traditional methods such as simulation and testing, formal methods allowed an earlier detection in the design cycle, thus reducing delays and costs.
- The code generation activity demonstrated the feasibility of LOTOS-based hardware/software co-simulation: by combining C code with LOTOS code compiled using

the EXEC/CÆSAR tool, a software emulation of an essential hardware component of POLYKID was obtained, which proved to function properly; the main limitation of the approach was a performance issue, which is likely to be solved by recent tool improvements.

- The test generation activity established that automatic test generation using the TGV tool was faster and more reliable than manual writing of deterministic tests; moreover, TGV gave better results than random test generation in terms of coverage and analysis of test execution. It was also established that the use of TGV would improve the current testing methodology by formalizing the concepts of testing.

From the industrial side, it was acknowledged that the proposed approach for system-level design was suitable for gaining a better understanding of the system, clarifying issues, detecting design mistakes, improving the overall quality of the product, and reducing the risk of delays. It was also acknowledged that formal methods could be introduced early in the design, even before reference specifications are stabilized.

As for related work, a number of significant experiments on the computer-aided verification of cache coherency protocols have already been published. In particular, we can mention the verification of the FUTUREBUS+ (a former IEEE standard, now withdrawn) protocol using the symbolic model-checker SMV [CGH+95], the verification of the SCI protocol using the MURϕ model-checker [SD95], the verification of the HAL S1 protocol [HFW97], the verification of the SCI protocol using the NUPRL theorem prover [FHS98], the verification of the FLASH protocol using the PVS theorem prover [PD98], the verification of the AVALANCHE protocols using the SPIN model-checker [NG98], the verification of lazy caching and snooping protocols using the MOCHA model-checker [HQR99], the verification of a snooping protocol using the symbolic model-checker VIS [SCJ+99], and the verification of SGI's ORIGIN protocol using SMV [Eir00]. A survey of verification techniques for cache coherency protocols can be found in [PD97]. Due to timing and cost constraints, it was not possible for us to compare the performances of these verification tools with those of CADP, as it would have required to model POLYKID in the various input languages used by these tools. A sketch of comparison between CADP and MURϕ can be found in [Che97]; yet, a rigourous assessment of different tools on the same cache coherency protocol remains to be done.

Our approach is novel in that it is not limited to verification: it also addresses other industrial needs, namely hardware/software co-simulation and testing. Moreover, we establish that these different activities can be performed in a coherent framework, using a common language and compatible tools. The CADP and TGV tools indeed provide a unique combination of useful features ranging from interactive simulation, embedded code generation, (compositional) model-generation, verification using bisimulations or temporal logics, test generation, and test execution. All these features are consistently integrated together, by means of the OPEN/CÆSAR environment notably, which increase the acceptance by industrial users. Recently, the integration was even made tighter, as TGV is now a fully-fledged component of CADP.

Finally, the experiment with POLYKID was found successful enough to launch a new BULL-INRIA collaboration, targeting at a new multiprocessor architecture currently devel-

oped by BULL. In a near future, we expect that formal methods and associated tools will become standard techniques for system-level design of complex systems.

Acknowledgements

The scientific achievements presented in this report are the result of a long-term BULL/INRIA project that spanned almost 3.5 years and involved many computer scientists and engineers, in three teams located in three different locations.

We would like to thank all those who contributed to this project and, first, the two former BULL project leaders: Nadia Tawbi, who initiated the project, and Ghassan Chehaibar, who played a significant role in the formal specification and verification tasks, but also:

- At BULL (Pregnana, Italy and Echirolles, France): Giuseppe Bosisio, Luca Bordoni, Luigi Casati, Paolo Coerezza, Margherita del Frate, Marc Derbey, Pierpaolo Maggi, Laura Populin, Ludovic Rattin, and Ferruccio Zulian;
- At INRIA Rhône-Alpes (Grenoble, France): Christophe Discours, Mark Jorgensen, Radu Mateescu, and Laurent Mounier (Verimag laboratory) for his help in using compositional verification tools;
- At INRIA/IRISA (Rennes, France): the TGV team of the PAMPA research group and particularly Pierre Morel (for the time he spent improving "on the fly" TGV during this experiment), Hakim Kahlouche (for developing the tester package) and Claude Jard.

We would also like to thank Fabrice Baray, Marc Herbert, Radu Mateescu, Frédéric Lang, Solofo Ramangalahy, and Nicolas Zuanon for their valuable remarks about this report.

Finally, we are grateful for the directors of DYADE, Eric Bantegnie, Gilles Bogo, and Patrick Valduriez for their continuous support.

References

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1988.

[CGH+95] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.

[CGM+96] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In Reinhard

Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, October 1996. Full version available as INRIA Research Report RR-2958.

[Che97] Ghassan Chehaibar. Use of Formal Methods in POLYKID Development. Available from `http://www.inrialpes.fr/vasy/dyade/polykid_1997.html`, October 1997.

[CSB00] Raul Composano, Andrew Seawright, and Joseph Buck. Modeling and Synthesis of Behavior, Control and Data Flow. In Egon Börger, editor, *Architecture Design and Validation Methods*, pages 1–48. Springer Verlag, 2000.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to Algebraic Specifications Based on the Language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[Dub89] Eric Dubuis. An Algorithm for Translating LOTOS Behavior Expressions into Automata and Ports. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*. North-Holland, December 1989.

[Eir00] Asgeir Thor Eiriksson. The Formal Design of 1M-gate ASICs. *Formal Methods in System Design*, 16(1):7–22, January 2000.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[FHS98] Amy P. Felty, Douglas J. Howe, and Frank A. Stomp. Protocol Verification in Nuprl. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer-Aided Verification (Vancouver, BC, Canada)*, volume 1427 of *Lecture Notes in Computer Science*. Springer Verlag, June 1998.

[FJJ+97] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, and César Viho. An Experiment in Automatic Generation of Test Suites for Protocols

with Verification Technology. *Science of Computer Programming*, 29(1–2):123–146, July 1997. Special issue on Industrially Relevant Applications of Formal Analysis Techniques. Also available as INRIA Research Report RR-2923.

[FJJV96] 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.

[FL93] 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 (Ottawa, Ontario, Canada)*, pages 305–312. North-Holland, April 1993. Revised version available from `http://lotos.csi.uottawa.ca`.

[FM91] Jean-Claude Fernandez and Laurent Mounier. A Tool Set for Deciding Behavioral Equivalences. In *Proceedings of CONCUR'91 (Amsterdam, The Netherlands)*, August 1991.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[Gar98] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[HFW97] Alan J. Hu, Masahiro Fujita, and Chris Wilson. Formal Verification of the HAL S1 System Cache Coherence Protocol. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors ICCD '97 (Austin, Texas, USA)*, pages 438–333. IEEE Computer Society Press, October 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture — A Quantitative Approach.* Morgan Kaufmann Publishers, 2nd edition, 1996.

[HQR99] T. A. Henzinger, S. Qadeer, and S.K. Rajamani. Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 301–315. Springer Verlag, July 1999.

[HT00] Ji He and Kenneth J. Turner. Verifying and Testing Asynchronous Circuits using LOTOS. In Tommaso Bolognesi and Diego Latella, editors, *Proceedings of the IFIP Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing, and Verification FORTE/PSTV'2000 (Pisa, Italy).* IFIP, Kluwer Academic Publishers, October 2000.

[IEE92] IEEE. Scalable Coherent Interface. IEEE Standard 1596-1992, Institution of Electrical and Electronics Engineers, 1992.

[IEE93] IEEE. Standard VHDL Language Reference Manual. IEEE Standard 1076-1993, Institution of Electrical and Electronics Engineers, 1993.

[IEE95] IEEE. Verilog HDL Language Reference Manual. IEEE Draft Standard 1364, Institution of Electrical and Electronics Engineers, October 1995.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[JM97] 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.

[KM97] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer Verlag. Extended version with proofs available as Research Report VERIMAG RR97-01.

[KVZ98] 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*, pages 211–226. Chapman & Hall, September 1998.

[LLGH89] 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.

[LSVS00] Luciano Lavagno, Alberto Sangiovanni-Vincentelli, and Ellen M. Santovich. Models of Computation for System Design. In Egon Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer Verlag, 2000.

[MdM88] J. A. Manas and T. de Miguel. From LOTOS to C. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 79–84. North-Holland, September 1988.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[NG98] Ratan Nalumasu and Ganesh Gopalakrishnan. Deriving Efficient Cache Coherence Protocols Through Refinement. In Dominique Mery and Beverly Sanders, editors, *Proceedings of the 3rd International Workshop on Formal Methods for Parallel Programming: Theory and Applications FMPPTA'98 (Orlando, Florida, USA)*, April 1998.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[PD97] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, 1997.

[PD98] Seungjoon Park and David L. Dill. Verification of Cache Coherence Protocols by Aggregation of Distributed Transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.

[PH97] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann Publishers, 2nd edition, 1997.

[SCJ+99] Srivatsan Srinivasan, Parminder Singh Chhabra, Praveen Kumar Jaini, Adnan Aziz, and Lizy K. John. Formal Verification of a Snoop-Based Cache Coherence Protocol Using Symbolic Model Checking. In *Proceedings of the 12th International Conference on VLSI Design "VLSI for the Information Appliance" (Goa, India)*, January 1999.

[SD95] Ulrich Stern and David L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Proceedings of the IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[Sjö91] Peter Sjödin. *From LOTOS Specifications to Distributed Implementations.* PhD thesis, Department of Computer Science, University of Uppsala (Sweden), 1991.

[ST93] Richard O. Sinnott and Kenneth J. Turner. DILL: Specifying Digital Logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Umit Uyar, editors, *Proceedings of the 6th International Conference on Formal Description Techniques FORTE'93 (Boston, MA, USA)*, pages 71–86. North-Holland, October 1993.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques – An Introduction to ESTELLE, LOTOS, and SDL.* John Wiley, 1993.

[YG98] Michael Yoeli and A Ginzburg. LOTOS-Based Verification of Asynchronous Circuits. Technical Report TR CS0951, Technion, Computer Science Department, Haifa, Israel, January 1998.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399