

5. Verification and modeling of Digital Systems (VDS)

Group Leader: L.PIERRE

(e-mail: Laurence.Pierre@imag.fr)

Members: D. Borrione, R. Clavel, L. Damri, L. Ferro, A. Helmy, K. Morin-Allory, Y. Oddos, A. Porcher (VDS/CIS)

Engineers: F. Pancher (part time), J. Sester

Interns: Z. Bel Hadj Amor, C. Deschamps, D. Enéas, J. Fischer

Research areas	Contracts	Industrial partners	Academic Partners
Verification Flow, Assertion-Based Verification, Synthesis of Hardware Devices from PSL Assertions, Formal Specification and Verification, Formal Methods for Fault Tolerance, Theorem Proving.	SFINCS (ANR), FME ³ (ANR), SoCKET (Pôles de compétitivité), Beyond DREAMS (MEDEA+), Nano 2012	Dolphin Integration, STMicroelectronics, Thales, EADS Astrium, Airbus	LIP6 (Université Pierre et Marie Curie, Paris), Radboud University (Nijmegen, NL), McGill University (Montréal, Canada), Royal Institute of Technology (Stockholm, Sweden)

The VDS group is mainly concerned by the *correctness of the hardware design*, from its early specification levels to the "register transfer" level (RTL). The research of the VDS group aims at providing *verification solutions based on formal or semi-formal methods*.

In both cases, the expected behaviour is expressed as a formal specification (roughly speaking, a statement in a formal language or logic). Formal verification makes use of static analysis techniques to prove that this statement holds for the hardware design, while semi-formal verification enables its verification during simulation. In the activities of the VDS group, formal verification is related to the application of deduction-based methods, through the use of theorem provers or proof assistants; semi-formal verification is identified with the construction and use of assertion checkers.

Depending on the size and the characteristics of the design under verification, one of these solutions is more advantageous than the other one. In particular, theorem proving is interesting for reasoning on very abstract (algorithmic) descriptions or for developing general-purpose theories (meta-models for general problems), and runtime verification using assertion checkers is beneficial for more concrete hardware descriptions of large size systems (for which static analysis methods such as model-checking are deficient). These aspects are developed in the following sections, where we describe our work on:

- the assertion-based verification and assertion-based design for IP blocks, using the notion of property checkers and test generators,
- the assertion-based verification of SoC's at the system level (described in SystemC TLM),
- a meta-model based solution for the formal verification of communication infrastructures in Networks on Chip (NoCs),
- the application of theorem proving techniques to the verification of robustness properties, using a formal representation of fault models.

5.1 Assertion-Based Design at RT Level

Members: K.Morin-Allory, Y.Oddos, D.Borrione, J.Sester, L.Pierre, L.Damri, F.Pancher, D.Eneas, C.Deschamps, J.Fischer

Assertion based verification (ABV) has raised a real interest over the past few years, and a large diffusion in the industry. Two IEEE standards have been defined to write temporal properties: PSL (Property Specification Language) and SVA (SystemVerilog Assertions). The main CAD tools available on the market provide the designer with furnished toolboxes combining ABV features with simulation: Questa, Verdi, NCSim etc.

Most often, ABV consists in synthesizing properties into hardware or software *monitors*. These components analyze the design, and detect any behavior violating the corresponding properties. More recent works have addressed the test vector generation from temporal properties. A property is synthesized into a component called *generator*. It produces signal sequences complying with the associated property. If the environment is described by a set of properties, an environment model can be obtained by combining the corresponding generators.

Our on-going works go beyond this concept, to achieve Assertion-Based Compilation (ABC). In this context, the following question is raised: is it possible to automatically produce a hardware module from a declarative specification composed of temporal properties? This problem is part of the following research domain: automatic synthesis of correct-by-construction circuits from specifications.

5.1.1 Automatic Compilation of Properties into Synchronous Monitors

The Horus tool¹ that we developed, based on formally proven correct methods, provides a unified support to assertion-based design, between the specification and the test phases. Given a set of logical and temporal properties written in PSL, Horus automatically constructs a test environment for the design. This construction is fast, correct, and produces efficient monitors and generators. Horus covers the whole PSL "simple subset", and the whole verification flow: from the simulation to the on-line testing. It can relieve the test engineer from the tedious work of writing correct test benches. When synthesized on FPGA, the instrumented design under test can execute at full speed.

The principles of the automatic compilation of PSL properties into synthesizable monitors have been described in previous reports. As a basic principle, the monitors are assembled following the syntactic tree of the PSL formulas, using as building blocks a library of primitive components. The library elements follow a systematic structure, and present a generic interface. The interconnection procedure is proven correct by induction, in higher order logic.

The status of each property in the instrumented design (strongly hold, hold, pending, fail), displayed on the *pending* and *valid* outputs of each monitor, can be traced at each clock cycle. In case of a failed property, the signals involved and the precise temporal origin of their sub-trace are identified. User-friendly post-processing debugging aids, such as graphical trace displays, are not available in our University prototype, but are planned in an industrial development in cooperation with Dolphin Integration.

Integration in SMASH and SLED

This Horus technology is now integrated in EDA tools of Dolphin Integration: SMASH (simulator) and SLED (schematic editor)².

The first release of Dolphin's mixed signal simulator SMASH integrating Horus-based support for PSL was SMASH 5.13.0 and was released in June 2009. Support for PSL was close to the full simple subset concerning FL operators, and only a few SERE operators were supported.

In SMASH 5.14.0 (released in December 2009), support for PSL sequences was improved: all operators are now supported, and the simulation model is generated instantly and is faster. SMASH 5.14.0 also introduces breakpoints on PSL properties which makes debugging a lot easier. Released in December 2009 as well, Dolphin's schematic link editor SLED 1.5.0 can create verification cells from PSL files, and makes it possible to instantiate and connect them to the design under test. It also customizes (i.e. renames signal names when necessary, which works around the lack of support for port mapping for vunits in PSL) the PSL verification units and generates appropriate SMASH directives when generating the netlist for SMASH, so that instantiated PSL properties are verified during simulation.

Additionally, the SLED SDG (Synthesizable Detector Generator) option, which makes it possible for the user to seamlessly generate PSL verification units as synthesizable VHDL or Verilog monitors, makes it possible to bring Assertion Based Verification into FPGA emulation, or to use PSL as a design language for safety related parts of critical applications.

An industrial case study has been used to assess this integration of the Horus technology in SMASH. The IP, provided as a RTL VHDL description, in an interface that implements the HDLC protocol. The HDLC (High-level Data Link Control) is a synchronous serial protocol used in data communications systems. The goal of the study was to analyze the protocol in order to express the specification (parts of the IP requirements) in PSL form. Thirteen properties related to the transmitter and the receiver parts of the

¹ <http://tima.imag.fr/vds/Horus/>

² http://www.dolphin.fr/medal/smash/evolution/smash_overview5.14.html and <http://www.carnot-lsi.com/nos-succes/dolphin-integration-integre-son-offre-une-technologie-de-luniversite-joseph-fourier>

interface were written and analyzed. This allowed validating and suggesting improvements of this HDLC design.

Rewrite Rules

This work has been performed jointly with Marc Boulé and Zeljko Zilic from McGill University, Montreal, Canada. The McGill group had developed their own checker generator tool, concurrently with our development of Horus. Comparisons between the two approaches showed similar results, despite the fact that the internal algorithmics are quite different: their work is based on automata and rewrite rules. A cooperation was established, with the support of a Jacques Cartier grant.

The use of rewrite rules was motivated by the richness of PSL. The rewrite rules were defined to convert various assertion forms into regular expression implications, for further use by simulation or formal verification engines. We have shown how an automated theorem prover can be used to ensure the correctness of assertion language equivalences and rewrite rules. We have implemented the higher-order semantic definition of PSL in the logic of PVS, and used the PVS proof assistant to mechanize the formal reasoning. This was applied to three sets of rewrite rules found in the literature, attempting to prove that they were well founded and semantically correct. We have shown that some of the published rules were actually incorrect. Of the more than fifty rewrite rules, two were not provable because of language semantics issues, nine were shown not to hold on empty traces, and four were shown to be false.

As witnessed in our proof results, we have also shown how certain simple subset guidelines must be changed in order to create behaviors that are better suited to dynamic verification with PSL. We have formally justified the guidelines for writing a simulation friendly PSL.

Low Power Monitors

In the first Horus prototype, and in most equivalent published works, the main figures of merit have been the speed and area of the synthesized monitors. If online checking is considered, monitors also have to be designed for power efficiency.

A new library of primitive monitors has been designed, using the clock gating principle, in order to reduce the dynamic power consumption. This technique is of interest only for the primitive monitors that contain 3 or more registers. The low power library has been proven equivalent to the original one, using conventional equivalence checking tools.

A set of benchmark properties has been synthesized in 350 nm and 65 nm, both for the standard and the low power libraries. The essential results are a set of curves showing the frequency of monitor activation for which low power monitors are beneficial in terms of power consumption. In all cases, a price must be paid in terms of area and frequency.

5.1.2 Test Sequence Generation for Accelerating ABV

The hardware monitors built using the Horus technology are used during simulation (or FPGA emulation) to check whether their representative properties are satisfied by the design. In that context, guaranteeing that the property is not verified *vacuously* is crucial. Generation of test sequences for the simulator cannot be performed purely randomly: test sequences must be designed to ensure a good coverage of the monitor's activation conditions.

Consider as a trivial example a property of the form `always (A -> B)`. If the testbench is such that `A` never holds, then the property is always verified, but verified *vacuously* i.e., without having actually been checked. The goal of this study is therefore to develop methods that can either determine that satisfying `A` is impossible, or produce test sequences that will force infinitely often the satisfaction of `A`. In the general case, properties under consideration may not be already in the implication form, and express requirements that involve primary outputs.

After studying existing solutions (ATPG methods, techniques for code coverage analysis, some results about the production of test sequences related to temporal properties,...), it was clear that none of them was actually adapted to our needs. We thus started the development of a specific method, first under the hypothesis that the circuit has already been synthesized as a netlist of gates and memory elements. In the general case, we transform the PSL property into a SERE (Sequential Extended Regular Expression) implication, using the rewrite rules mentioned in the previous section, then we focus on the left hand side of this implication. Each "letter" of this SERE has to be characterized in terms of the primary inputs and internal registers. To that goal, an algorithm has been specified and implemented; it has been tested on some simple examples. Obtaining such a characterization will allow the application of our Horus test generators to this test sequence generation issue.

5.1.3 Automatic Compilation of Properties into Synchronous Generators

In contrast to *assertions* that are turned into monitors to check if the design is compliant with the corresponding property, *assumptions* are used to constrain the behavior of the environment, i.e. restrict the set of possible input vectors. By not considering test vectors that are not compliant with the design input specifications, the test can be more efficient. Modeling the environment of the design with formal assumptions is recent. In this research, we focus on this specific problem: automatically build a simplified model of the environment that feeds the design under verification with test vectors that satisfy a set of assumptions. This model is a (test vectors) *generator*.

Two versions of the compiler of assumptions into synchronous RTL generators have been implemented:

- A modular construction based on the principles of the monitor compilation. The elaboration of this first prototype was described in previous reports.
- An automata-based construction, built according to the principles of the MBAC tool from McGill University. This second prototype, developed in 2009, is now described.

Generators and monitors are symmetrical concepts: a monitor recognizes the language of a property (actually its complement) while a generator produces the language of the property. It is thus a natural idea to use the same central automaton construction technique to produce property monitors and generators. Since the monitors built by MBAC are of low complexity, we decided to reuse the MBAC approach and automaton construction mechanism to produce synthesizable generators.

Figure 5.1 illustrates the two steps required to build a generator from a temporal property. First the MBAC tool elaborates the checker's automaton. Then the MBAC_GEN module turns it into a generator automaton that describes all the traces compliant with the corresponding property. Finally the back-end tool extracts the synthesizable HDL description for the generator. In particular, as several traces can satisfy the property, more than one path may lead to an accepting final state: a pseudo-random block is used to randomly choose one transition in the states that exhibit non-deterministic transitions.

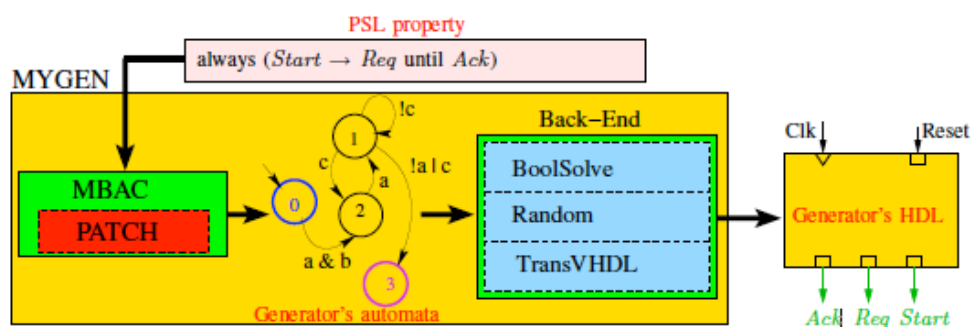


Figure 5.1 The generator's construction flow

A benchmark of 60 generators has been synthesized with Quartus II 7.2 on a Cyclone II EP2C35 FPGA. Measures taken after synthesis have shown that, in contrast to the generators produced by the first prototype, the complexity is related to the automaton, but not to the property. It is then very hard to forecast the size of the generator from the structural complexity of the property: which Boolean and temporal operators are used is a prevalent criterion.

5.1.4 Automatic Compilation of Properties into Synthesizable Designs

Combining the monitor and generator viewpoints above, one arrives to Assertion Based Compilation (ABC). A property is seen as the specification of the module to be designed, where some operand variables are input to the module, and others are outputs. The objective is then to directly produce the RTL design from its assertions.

Following the previous approach, the compilation method is modular: it is based on the interconnection of elementary library modules for the simple PSL operators, according to the syntactic structure of the property. For ABC, we introduce a third type of component: the *reactant*. A reactant is a RTL design automatically generated from a specification written under the form of an asserted temporal property. A reactant has

inputs and outputs that are operands in the PSL formula that defines the property at hand; it reacts to the input values and produces output values so that the property holds. The construction method is again based on the proven correct interconnection of proven correct elementary reactants, but also elementary monitors and generators.

One reactant is produced for each property. In the general case, the specification has more than one property, and a same variable may appear in several distinct properties. A difficult problem to be solved is the elaboration of a method, and its formal justification, for deciding which properties monitor the variable, and which properties generate its value. Moreover, if a variable is an output for several reactants, these are combined with a special kind of component named solver, to produce the final design.

Our ABC method has been implemented in a prototype software tool called SyntHorus. The compilation complexity is linear in the number of operators of the specification. SyntHorus can process complex specifications that hold hundreds of properties, and produces the final circuit in a few seconds. The size of the resulting RTL circuit is also proportional to the size of the specification.

FPGA experiments have been done on complex circuits: crossbar controller, GenBuf, Net-Maker etc. They showed that automatically synthesized designs are globally more complex than the corresponding hand coded ones. We have demonstrated that the overhead is directly linked to the number of multi-source signals present in the specification. In their absence, no hardware resolution mechanism needs to be embedded and the resulting design is slightly larger than the original one.

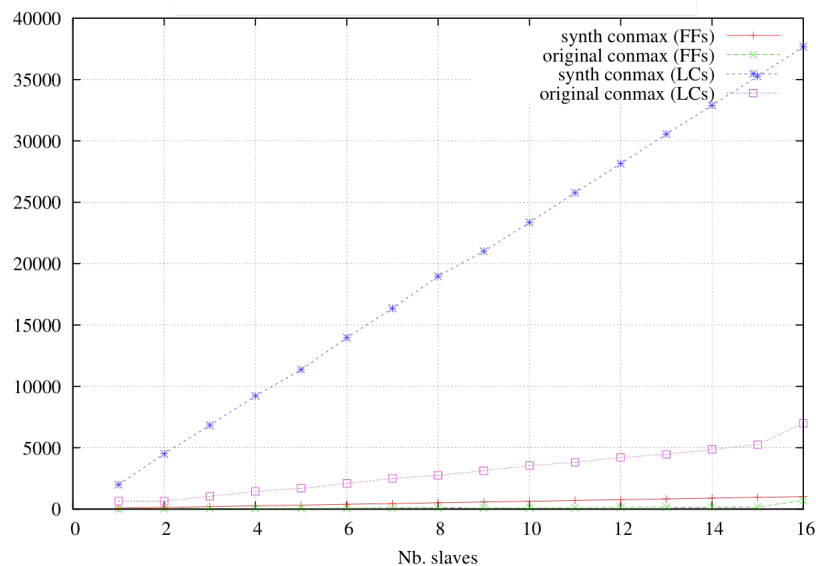


Figure 5.2 Comparison of synthesis results on FPGA between an original Conmax IP and the model compiled with SyntHorus

As an example, **Figure 5.2** shows the comparison between a manually designed Conmax IP and the automatically generated design with SyntHorus, for 4 masters, using two priority levels, connected to a number of slaves varying from 1 to 16. LCs overhead is clearly visible and remains linear when the number of slaves increases. The main part of this overhead is due to the use of the Solvers. The FFs overhead is noticeably less important.

Our current works aim at removing the weaknesses of the first SyntHorus prototype, both in terms of area efficiency, and with regards to the currently non-automatic determination of the (observed/generated) direction of the internal signals.

5.2 Assertion-Based Design with an Asynchronous Technology

Members: A.Porcher, K.Morin-Allory

The work reported here is realized in cooperation with the CIS group. Similarly to synchronous monitors, asynchronous monitors can be employed for logical simulation or emulation onto a FPGA board. While in synchronous circuits a clock globally controls the activity, the asynchronous circuit activity is locally

controlled using communicating channels that detect the presence of data at their inputs and outputs. This is consistent with the so-called handshaking or request/acknowledge protocol. One transition on a request signal activates another module connected to it. Therefore, signals must be valid at all times. Asynchronous circuit synthesis must be hazard- free. In order to have very reliable monitors, we choose to implement Quasi-Delay Insensitive (QDI) circuits. Indeed, these circuits are very robust to Process, (strong) Voltage and Temperature (PVT) variations. Moreover, they offer nice properties such as modularity and low-power consumption.

Like in the synchronous case, the monitor construction mimics the structure of the property. It is based on a library of primitive monitors (one for each PSL operator) and on a syntax directed interconnection scheme. **Figure 5.3** illustrates the construction of the monitor for property:

PSL property P1 is always A -> (B before C);

A monitor observing a synchronous design takes as inputs the Reset, the Clock and the signals (A, B and C) of the design that are operands in the property. The monitor outputs the signal Valid (coded in dual-rail) that provides the evaluation result ('1' on the second rail means error, otherwise it means absence of error). To comply with the handshake protocol, the signal Valid is acknowledged. To avoid any behavioral modification in the design under verification, the observed signals and the synchronization signal are not acknowledged. Each synchronous observed signal is turned into a signal that respects the 4-phases protocol via a Synchronizer. This synchronizer has been formally proven correct and fully characterized.

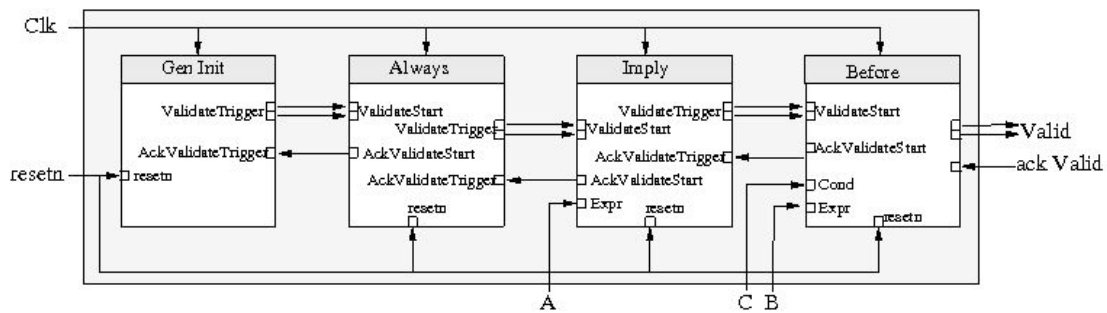


Figure 5.3 Asynchronous monitor for property P1

Monitors have been implemented on both an ALTERA DE2 EP2C35 FPGA board and on a library of standard Muller cells in a CMOS 65nm technology from ST Microelectronics. As it is shown in the following Table, the monitor area is very small and so induces a very low over-cost for monitoring an integrated system, Furthermore, the area grows linearly with the complexity of the asynchronous monitor.

PSL P roperty	Area (µm ²)	Freq. (GHz)
assert A	241,00	1.85
assert A Until! B	570,00	1.25
assert Always A	486,00	1.66
assert A Before! B	528,00	1.35
assert Next[2] A	549,00	1.66
assert A -> (Next B)	521,00	1.66
assert A or (Next! B)	573,00	1.66
assert Next a[3..7] A	1769,00	0.78
assert Next e[3..7] A	2310,00	1.04
assert Next Event!(A) B	553,00	1.42
assert Next Event(A)[3] B	1272,00	1.25
assert Never A	355,00	1.31
assert Eventually! A	427,00	1.61
assert Next Event a(A)[1..3] B	1496,00	1,00

5.3 Assertion-Based Verification at System Level

Members: L. Ferro, L. Pierre, Z.Bel Hadj Amor

With the increasing SoC complexity and time-to-market pressure, *platform-based design* is becoming a new paradigm for the design and analysis of embedded systems. Among its advantages, this methodology favors design reuse, architecture exploration, and early software development. It allows to raise the abstraction

level to ESL (Electronic System Level), thus improving the efficiency of various activities such as validation. In that ESL context, *SystemC TLM* (Transaction-Level Modeling) is perceptibly being adopted, and TLM specifications tend to become golden reference models. Their reliability is therefore capital, and guaranteeing their correctness is compelling.

To that goal, we have developed a solution for the *runtime verification of PSL assertions for TLM specifications*. At that level, assertions express properties regarding communications i.e., properties associated with transactional actions (for instance, data are transferred at the right place in memory, a transfer does not start before the completion of the previous one, etc.). Our verification solution makes use of: (1) a method for the construction of SystemC checkers (monitors) from PSL assertions which is an adaptation of the Horus technique, and (2) a framework for the observation of communication actions. This solution has been implemented in a prototype tool with a graphical user interface, ISIS³. PSL assertions can be captured through the GUI and:

- the tool performs the automatic construction of the corresponding monitors,
- then the SystemC code of the design under verification is instrumented to relate the variables involved in the properties (formal variables) to actual components of the design.

The monitors are thus linked to the design under test through the observation mechanism, and it remains to run the SystemC simulator on the system made of this combination of modules. Any property violation during simulation is reported by the monitors.

Through its Modeling layer, PSL gives the possibility to use *auxiliary variables* in assertions. To illustrate the usefulness of this feature at the transactional level, let us consider the example of a Motion-JPEG decoding platform⁴. The following property *P* can be used to check the reliability of its communication channel: *the data that are displayed (written on the RAMDAC) are exactly the ones that have been transmitted by the EU (processing unit)*.

The expression of this property requires the memorization of the data that are transmitted by the EU, to be compared to the data subsequently written on the RAMDAC:

```

/* Modeling layer */
unsigned int req_data;    // auxiliary variable
// When the EU transmits data, they are stored into req_data:
if (eu.write_CALL())
    req_data = eu.write.p2;
/* Assertion */
// Every time the EU transmits data, the data subsequently written on the RAMDAC
// will be identical to those previously transmitted data:
assert ALWAYS (eu.write_CALL() ->
               NEXT_EVENT(rdac.write_CALL())(req_data == rdac.write.p2));

```

The Modeling layer allows the declaration of auxiliary variables. In the statement part, statements of the underlying language (C++ here) can be used to initialize and update these extra variables. Here *req_data* is a variable that memorizes the transmitted data; it is updated when *eu* executes its method *write*, the term *eu.write.p2* denotes in ISIS the second parameter of function *write*. The assertion states that, each time *eu* writes in the communication channel then, next time data will be written on the RAMDAC, these data will be identical to *req_data*. Memorization is mandatory because the moment when the comparison *req_data == rdac.write.p2* is performed is concomitant to *rdac.write_CALL()* but subsequent to *eu.write_CALL()*.

To implement this notion in the ISIS tool, we have proposed a formal (operational) semantics of PSL endowed with the Modeling layer, and we have refined it to fit the transactional level.

It is worth noticing that, in property *P* above, only one auxiliary variable *req_data* was necessary. Indeed, the communication channel does not process several communications simultaneously. Therefore we memorize the transmitted data when the communication starts, and we can use this value to check the data written on the RAMDAC when it ends.

In the case where *several communications can occur concurrently*, using only one global auxiliary variable is no more sufficient. Several verifications of the same assertion, for different values of one or more variables, can overlap on the same evaluation cycles. In other words, we need to enable reentrancy for the PSL assertions.

Therefore we have proposed, and implemented in ISIS, a solution to support reentrant assertions (i.e., different instances of a same assertion evaluated on overlapping evaluations cycles), through the use of multiple checker instances, with local variables. We have extended the PSL syntax with an appropriate syntactical construct, and we have adapted the semantics accordingly.

³ <http://tima.imag.fr/vds/isis/>

⁴ provided by the SLS group

5.4 Formal verification of NoC communication infrastructures

Members: A. Helmy, L. Pierre, D. Borrione

The current technology allows the integration on a single die of complex systems-on-chip (SoC's) that are composed of manufactured blocks (IP's), interconnected through specialized networks on chip (NoCs). IP blocks have usually been validated by diverse techniques (simulation, test, formal verification) and the key problem remains the validation of the communication infrastructure. The work reported in this section addresses the *formal verification of NoCs by means of a mechanized proof tool, the ACL2 theorem prover*.

In the context of platform-based design, the trend in the design flow is to raise the level of abstraction of the initial phase and to ground the flow on verified parameterized library modules. Yet, on-chip communications are not supported by a general and formal theory, which is necessary to obtain verified parameterized communication modules. Our objective was to provide a *formal foundation to the verification of on-chip communication networks*, spanning from their initial design specifications to their RTL implementation.

As a first step toward this objective, we proposed the generic network on chip model (GeNoC)⁵. It consists of a metamodel of on-chip communication architectures and its implementation in the logic of a theorem proving system. The peculiar aspect of this model is to represent a large class of systems. It is a highly generic and parameterized object. While performing the proofs, parameters such as the size of the network or the length of messages need not to be instantiated. Its implementation in a theorem proving system provides mechanized support and partial automation in the verification effort. The model is implemented in the ACL2 theorem prover. An important feature of ACL2 is to denote both a powerful theorem proving system and an execution engine in the same environment. The theorem proving system has a high degree of automation. ACL2 specifications are written in an applicative subset of Common Lisp and are thus executable.

Our first versions of the model contained unrealistic simplifying hypotheses, concerning the granularity of moves and the time when messages were introduced in the network. As a consequence, while the final result of the communications was correctly portrayed, the intermediate steps were abstracted away: where and when a message is temporarily delayed by the presence of other messages in the network could not be shown. The enhanced formalization proposed as a second step constitutes a significant progress, both mathematically simpler and offering a much larger expressive power:

- (i) the current metamodel covers a network specification from the *transport* layer to the *data link* layer of the OSI model;
- (ii) the progression of the messages in the network is specified step by step instead of considering their transfer atomically from their source to their destination, thus allowing a great variety of scheduling policies (circuit, wormhole, priority, etc.);
- (iii) new messages may enter the model at arbitrary times and arbitrary nodes;
- (iv) the same model can be used for formal verification and for simulation.

This metamodel represents the transmission of messages on a *generic* communication architecture, with an *arbitrary* network characterization (topology and node interfaces), routing algorithm, and switching technique. The main function of this model, called GeNoC, is recursive. Each recursive call represents one step of execution, where messages progress by at most one hop. Such a step defines our time unit.

A first *correctness theorem* is associated with function GeNoC. It states that for all topology T , interfaces I , routing algorithm R , and scheduling policy S that satisfy specific constraints P_1 , P_2 , P_3 and P_4 , GeNoC fulfills the following correctness property: *every message arrived at some node n was actually issued at some source node s and originally addressed to node n , and it reaches its destination without modification of its content*.

The proof of this correctness property is derived from constraints (proof obligations) P_1 , P_2 , P_3 and P_4 , without considering the actual definitions of the constituents. Consequently, the global correctness of the network model is preserved *for all particular definitions* satisfying the constraints. Following the same principles, a second meta-theorem has been proven, it is concerned with the fact that there is *no loss of messages* under given constraints (proof obligations).

Proving that a particular NoC is a valid instance of GeNoC amounts to:

- defining the functions that describe the network topology,
- defining all the functions associated with the communications,
- discharging the proof obligations for these functions.

⁵ <http://tima.imag.fr/vds/GeNoC/genoc.html>

This technique has been applied to various NoCs:

- the Hermes NoC (PUCRS, Brazil): <http://www.inf.pucrs.br/~gaph/Projects/Hermes/Hermes.html>,
- the Spidergon NoC from STMicroelectronics,
- the Nostrum NoC (Royal Institute of Technology, Stockholm): <http://www.ict.kth.se/nostrum/>.

5.5 Application of formal methods to fault-tolerance

Members: R.Clavel, L.Pierre

The work reported here is performed in the framework of the FME³ project, in cooperation with the ARIS group⁶. Designing dependable circuits requires in particular evaluating, at each step in the design flow, the achieved level of robustness against various types of faults or errors. In critical systems, an erroneous piece of information may lead to dramatic consequences in terms of human lives. In those cases, errors are generally the consequence of natural phenomena such as particle impacts, electromagnetic perturbations, electrical noise or degradations due to aging. The causes of errors, called faults, were usually modeled in digital systems as single bit-flips or signals stuck either at 1 or at 0. With the evolution of technologies, circuits are increasingly sensitive to transient faults that have therefore become the main concern for designers. Also, faults increasingly lead to multiple-bit errors that are more difficult to detect or tolerate in the system. We target the development of new methodologies for *analyzing the robustness of circuits described in VHDL at the Register Transfer (RT) level*, with respect to errors caused by transient faults. Our goal is to take advantage of formal methods to get accurate and efficient solutions that would complement existing fault-injection techniques.

In a first solution, we intended to take advantage of the high level of automation of the ACL2 theorem prover. We defined and then formalized in ACL2 the fault model that corresponds to the presence of a single or multiple-bit error in a single register of the circuit. This model characterizes the fault-injection function *inject* as a function that satisfies the following conjunction of properties:

- it takes as parameter a state $s \in S$ and returns a state $\text{inject}(s) \in S$
- $\text{inject}(s)$ is different from s
- only one memorizing element (n -bit register) differs from s to $\text{inject}(s)$.

In other words, the injection function belongs to the following set:

$$E = \{\text{inject} : S \rightarrow S \mid \forall s \in S, \exists ! i, (\text{inject}(s))_i = f(s_i)\}$$

Encoding such a formulation in a theorem prover would require the presence of quantifiers and the possibility of specifying functions by characteristic properties instead of using a function definition. Despite the fact that ACL2 is first-order and does not support the explicit use of quantifiers, there are solutions to mimic certain kinds of originally higher-order or quantified statements. For instance, the encapsulation mechanism allows to introduce new function symbols that are constrained to satisfy certain axioms, without providing function definitions that uniquely determine the functions's behavior. Using this principle, it was possible to encode the model above in ACL2. However the main problem with this implementation is not the use of the "encapsulate" construct, but the fact that the implementation of the third property ("only one state component differs from s to $\text{inject}(s)$ ") in the error model had to be expressed by a theorem that explicitly enumerates every possible error location, thus leading ACL2 to enumerate all the corresponding subgoals. CPU times could become prohibitive with real-size systems. The conclusion was that it is possible to find a way of encoding our models and associated robustness properties in ACL2, but that it may result in a lack of efficiency for performing proofs.

We thus decided to adopt a less automated proof tool, that provides a more powerful logic. One of the candidates was PVS. Its logic and inference mechanisms are powerful enough, without requiring too much effort for realizing proofs. We can take advantage of several PVS features:

- the possibility to parameterize PVS theories,
- the possibility to define predicate functions that make use of universal and existential quantifiers,
- the possibility to override a function definition, by means of the WITH construct. The result of an overridden function is exactly the same as the original, except that at the specified arguments it takes the new values.

The fault model was encoded in PVS, thus making explicit the fault function f , and enabling the

⁶ <http://tima.imag.fr/vds/FME3/>

characterization of a parameterized set of injection functions:

$$E(F) = \{\text{inject} : S \rightarrow S \mid \forall s \in S, \exists i, \exists f \in F, (\text{inject}(s))_i = f(s_i)\}$$

where F is a set of fault functions. Choosing different instances for this parameter F easily allows considering the same circuit with several fault models for its registers.

We also defined an extension of this model to characterize the presence of a single or multiple bit error in several registers of the circuit.

These formalizations and the corresponding representations of robustness theorems were applied to examples such as a cash withdrawal system, a FIR filter, and a CAN interface.

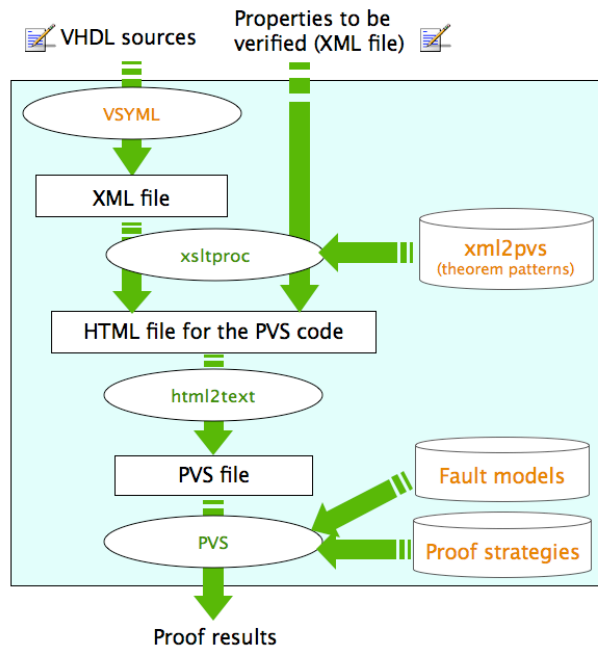


Figure 5.4 VHDL to PVS toolchain

A toolchain has been created to automatically generate the PVS source code from the original VHDL RTL description (see **Figure 5.4**). Using our tool VSYML, we parse the original VHDL RTL code, perform symbolic execution, and we get an XML representation of the transition and output functions. Templates of robustness theorems have been encoded in a library, `xml2pvs`. The intermediate format produced by VSYML is processed, together with the properties to be verified and the patterns of `xml2pvs`, by the tools `xsltproc`⁷ and `html2text`⁸ in order to produce a PVS file that contains both the design description and the theorems to be proved in the presence of soft errors. Proof strategies dedicated to the various types of robustness properties have been defined. They enable the mechanization of the proof process.

⁷ <http://xmlsoft.org/XSLT/xsltproc2.html>

⁸ <http://www.aaronsw.com/2002/html2text/>