On the design of a Formal Debugger for System Architecture

Bruno Monsuez
ENSTA - UEI
32 bd Victor
F 75739 Paris Cedex 15
email: bruno.monsuez@ensta.fr

Franck Védrine
CEA, LIST
Software Reliability Lab, Boîte 65
CEA Saclay, Gif sur Yvette,
F-91191 FRANCE
email: franck.vedrine@cea.fr

Nicolas Vallée
ENSTA - UEI
32 bd Victor
F 75739 Paris Cedex 15
email: nicolas.vallee@ensta.fr

Abstract: System level design helps alleviate much of the burden of usual RTL level design except verification. Functional verification of system architecture may first seem easier at the higher level of abstraction. In fact, insight on fundamental aspects of semantics of system level design and impact on lower levels of abstraction makes it much harder. Formal verification has received till now little attention for functional verification of system architecture. However, formal verification may provide great benefits to the system level designers. In this paper, starting from the experiments we have conducted with analyzing and verifying SystemC models using formal verification techniques, we describe what a potentially very useful approach to build a formal debugger to verify system architectures designed using a modern system model language like SystemC or SystemVerilog.

Key–Words: Formal debugger, SystemC, Hypergraphs, Abstract interpretation

1 Introduction

Functional verification is widely recognized as the major bottleneck of the hardware design industry. The ever-growing demand for performance and time to market combined with the exponential increase in hardware size as well as in system complexity cause the verification task to become increasingly difficult. Leading chip and system companies with almost infinite resources still don't arrive at bug free designs in a short and predictable design cycle.

Functional verification of a hardware design after completion of the design is extremely costly and cannot be achieved in a limited time. The way that is currently explored by all the major chip and system companies is first to develop and test an abstract model of the system architecture, for instance a transactional level model, and test the functionalities, reliabilities and performances if such a model allows it.

System modeling seems the most natural way to go to ensure an efficient and functional system architecture. Current development in the HDL language arena emphasizes on system modeling languages like SystemC[1], SpecC[2]With the introduction of the new system modeling languages, a set of new tools have been also introduced for testing and making system simulation less time consuming,. For instance, special verification languages like Vera[3], Verisity's [4] and Cadence's SystemC Verification Library [5], have been developed to support automatic stimulus generation. What the industry expects from system modeling and extensive testing of system architecture is to uncover bugs early and to achieve high quality of the design.

However, verifying the system architecture at a transactional level [10] requires a lot of computation time and a good expertise in their design. The complex use of the powerful verification languages requires a high level of expertise to be profitable.

Formal verification has already been successfully used for proving and verifying hardware design[6][7]. However, current available tools cannot be used to achieve functional system architecture. Since system level verification is quite new and till now, there was little industrial experience with functional verification of system models at transactional level, it was first difficult to determine if formal verification technologies could be successfully used for their verification and secondly there was not enough feedback to identify the requirements that such a tool should fulfill.

For three years, we cooperate with STMicroelectronics to introduce and evaluate the benefit of formal verification for SystemC model verification. We successfully experiment with Bus Cycle Accurate design as well as with transactional level models.

In this paper, starting from our experience, we expose what are the expectations of a system designer with respect to a debugging tool for system level architecture that is based on formal methods, what are the benefits of using such a debugging tool during the design of a system architecture and we conclude with a quick overview of the architecture of a formal system level debugging tool that has been labeled by and

received support from RNTL[8] and that will offer the previous identified capabilities.

2 Formal Verification of System Architecture

Functional verification of system architectures and unitary component verification have little in common. The first operates at the very first design stage. IPs are described at an abstract level, precise information like time and clock are missing, the designer concentrates on computational and transactional behaviors of the individual components.

The goal of this design state is to verify that the block architecture as well as the communication or interaction between the identified components may be functional. There is at this stage no guarantee that the final implementation of the low-level conforming components will be bug-free. However, if the behavior of the low-level components doesn't conform to the behavior of the abstract components, it is quite sure that the required functionalities and performances wouldn't be matched by the implementation.

Successfully tested and verified architectures provide "golden models". Implemented components should adhere to the abstract behavior of the abstract components used in the system architecture model.

2.1 What can we expect from a formal debugger ?

To deliver meaningful results, a formal debugger must allow the designer to verify and to extract properties about: (1) the abstract behavior of the components, (2) the protocols of the global system as well as the protocols specific to an individual component, (3) the invariants on the global system or on a group of individual components, (4) the simulation/execution traces, (5) the data and values that are sent or received by the various components during simulation.

To succeed, a formal debugger must provide support: (1) for defining or extracting assertions or assumptions, (2) case-reasoning, (3) formal reasoning, (4) as well as static analysis.

2.2 What are the used verification technologies ?

The current methodologies used by architecture verification largely depend on simulation. Industrial solutions like [9] automatically generate precise and complete testbenchs. The testbenchs and their associated oracles supply assumptions and hypotheses that the implementation of the components should verify. Simulation-based techniques also supply basic functionalities to find the origins of errors and to correct them through debugging.

Formal methods can deliver conformance verdict. Despite recent progress, they're still limited to small design blocks or small systems. Theorem proving are used to verify some specific parts of processors involving complex and arbitrary reasoning based on induction proofs [12]. Model checking is adequate for small designs (up to 500.000 gates) when the inherent model is finite and description and specifications may be expressed within the logic supported by the model checker. Property inference defines purpose-oriented tools with efficient verification on large design. However, the inferred properties are very specific. Abstract interpretation automatically finds semantic properties on the system but may loose a lot of precisions if the abstraction domains aren't adequate.

All those formal methods complement themselves: theorem provers assist the designer in synthesizing hypotheses and assumptions that should be verified. Abstract interpretation and property inference can automatically infer additional and pertinent properties. In some cases, the abstract interpretation may also deliver a conformance verdict. Finally model checking can deliver a conformance verdict: either the system verifies the assumption or the system violates one assumption and the model checker gives an execution trace that leads to the violated assumption.

Distinguing the different technologies matters when we know how inferring and extracting complex assumptions or hypotheses on real systems is.

3 Designing a formal debugger

Using "static analysis" in a formal debugging isn't new. Bourdoncle for instance has developed the "syntox" tool[18] based on abstract interpretation to analyze and debug programs written in "pascal".

However implementing tools for the formal debug of a high abstraction design IP is as much a technical as a technological challenge because of (1) the need to have a tool usable by everyone who is not a specialist in formal verification, (2) the need to have a very fast tool that still provide valuable information.

The classical approach, called "design and debug" cannot be used in the case of exhaustive verification. Implementing the components, verifying the components, correcting and verifying them again make verification a time consuming job. If we add the fact that the currently available tools require a high expertise in dynamic or formal verification, this process makes the use of such tools unsuitable for the industrial chip designer. Therefore, we decided to take a different approach, we integrate the debug/verification process during all the development stage of the architecture, providing a platform that is similar to a standard debugger but that manipulates "formal properties". We called this approach "debug as design".

3.1 A common debug platform

A full automatic analyzer or a push-button analyzer will only do a few incomplete verifications. Too many warnings may be also produced and it will be very difficult to figure the origin of each warning, preventing any certification. The analyzer must be associated with a formal execution tool and a property extraction tool driven by the user. These tools must be capable of pointing out any error level to the user as well as distinguing sure errors and potential errors.

Moreover, we list in the following standard requests that the user may issue to explore the results of the analysis.

- the user may request an "erroneous scenario" to find the origin of a possible error, or a "custom scenario" to ensure the good execution of the process and determine the invariants.
- computes and browses all scenarios that lead to a control point, starting from this control point.
- asks for the value of a given register at a given time or asks for the value boundaries of a given register during a given period of time,
- asks simple questions like: is the value of one variable higher than the value of another one ?

To provide a viable platform that allows the development and the use of formal verification during the design of system architecture at transactional level, we identify the additional four key requirements:

Modularity: Formal verification of system architecture must support the analysis of a group of components in the absence of the non-available components. It must also support replacing a component without having to analyze the complete system again.

For instance, a designer should be able to start the verification process of an MPSoc design with for instance two processors and a very abstract description of the components that ensure cache coherency of both processor's cache memory. In a second phase, he will certainly connect both processors to an external bus that communicates with a shared memory. If formal verification supports modular and context-independent analysis, the designer will be able to start the verification process with the very simple architecture where the two processors and the components that ensure cache coherency are defined. Designer can ensure that the basic architecture is correct and then adds new modules or replaces current modules with new ones without having to recompute all properties inferred or verified by the previous model.

Heterogeneous analyses: When designing the system architecture, the system designers are interested in validating key aspects of its system architecture. According to his work, he will use heterogeneous abstraction levels; some IPs will be pure transactional models, some other IPs mixing transactional levels and Bit & Cycle Accurate models and finally some IPs pure BCA models. For a given abstraction level, there is also miscellaneous aspects that may be modeled. For instance, exploring the transactional level modeling, Cai & Gajski [10] distinguishes between specification, component-assembly, bus-arbitration, bus-functional and cycle-accurate computation models.

Finally, properties to be analyzed such as timing, protocol, data flow also strongly depend on the abstraction level of the architecture components. A formal verification platform may be able to handle all the abstraction levels as well as the different notions and properties that the system designer wants to model.

A verification at low cost: Everyone agrees to say that verification consumes 80% of conception time. The implementation of architectures, environment for high-level test as well as formal verification of high-level IP components already consume over 25% of total time for the verification process of a component. The more features and possibilities the systems offer, the higher the conception time ratio will be. It is important to obtain a low cost architecture implementation. Since system architecture modelization is a dynamical part of the development and is open to a fast evolution, verification must not be an impediment to the flexibility of this stage.

Assembling the technologies: A final challenge but not the more difficult one, is to assemble the most pertinent technologies in development, verification and static analysis. In section 2.2, we point out theorem proving, model checking, property inference, abstract interpretation, simulation among others. Those technologies successfully and efficiently apply to a given set of properties as well as to a given abstraction level, but there is by now no technology that applies to all the properties and possible abstraction levels. We hold the idea that we must use and combine several technologies so that we may cover more properties at the lowest possible cost. Assembling all results make possible to refine the results gained by the use of one technology. Again, combining different technologies may also in some case speed up the analysis too.

However, developing a formal debugging platform where you can combine static analysis based on model checking, property inference, abstract interpretation as well as second order analysis is new.

3.2 The technological challenges

Currently for each formal verification techniques a representation is tailored for the given application. Since precision and efficiency of the analysis heavily rely on the program or system representation, designers define the best suited one for the given technique.

The first identified drawback is that when combining more than one formal verification technique, complex utilities must convert one representation into another one. Such conversions introduce a lot of work and may sometimes introduce loss of information.

The second identified drawback is that since there is no common representation for system and properties extracted from it as well as properties that it must verify, it is quite difficult to combine analyses that rely on different techniques. In most of applications, the analyses are conducted in parallel and their results are combined. However, there are no interaction between the analyses and no mutual-overcrossing which may lead in a much more precise results for each analysis.

The third identified drawback is : since in the "debug as design" approach the debug/verification process works all along the development stage, the incremental debug and verification results should be preserved and combined with the results obtains after the instantiation or the addition of components to the architecture, that means *the formalism used to express the system to be analyzed should be the same that the formalism used to express the results of the analysis*.

Consequently, a unified representation must support (1) mixing different abstraction levels and must support refinement ; (2) mixing specifications and code, it must also support mixing extracted properties and code ; (3) program code or hardware description and must also support a representation of program execution or system simulation ; (4) assembling modular and parametrized components ; (5) finally allowing to mix hardware and software designs since there are an important and growing clas of co-designs in which hardware and software are tightly coupled.

3.2.1 1st: same structure for codes, properties and data

To implement a formal debugger for system architecture – this approach can also be used to implement formal debuggers in general – we propose an unified representation based on a well-known structure called hypergraph. Starting from this structure we extend it to a more advanced structure that we decided to call *recursive or fractal hypergraph*. Hypergraphs are a mathematical extension of graphs : a graph whose hyperedges may connect two or more vertices. However undirected structures are of little use for modelling program execution of system behaviors. We are mostly interested in directed hypergraphs.

Definition 1 (Directed hypergraph) A directed hypergraph $\mathscr{H}$ can be defined as a pair $(\mathscr{V}, \mathscr{E})$ where $\mathscr{V}$ is a set of vertices and $\mathscr{E}$ is a set of directed hyperedges. Each hyperedge e is a pair of two set of vertices: $\mathscr{E} = \{(\{u_{\text{in}}, v_{\text{in}}, \ldots\} \times \{u_{\text{out}}, v_{\text{out}}, \ldots\}) \in (2^{\mathscr{V}} \times 2^{\mathscr{V}})\}$

Directed edges of a graph modelize a possible transition between an initial state and a final state. Directed hyperedges of a hypergraph modelize one or more transition connecting a set of initial states to a set of final states. Thus, the hyperedges denote any system that connects a given set of initial states to a set of final states. An interesting subset are the systems that can also be represented using a graph. For instance, automata's and other graph based representation like petri nets define a transition system that maps a set of initial states to a set of final states. They could be abstracted by a hyperedge that maps the initial states to the final states. We would like to generalize this notion to hypergraphs, restricting the hyperedges to hypergraphs. To quickly summarize, fractal hypergraphs are hypergraphs where the hyperedges between the vertices are defined by hypergraphs too.

Definition 2 (Basic fractal hypergraph) A basic fractal hypergraph $\mathscr{HB}$ is defined as :

- a set of vertices $\mathscr{V}$,
- a set of sub-fractal hypergraphs $\mathscr{H} = (h_i = (\mathscr{V}_i, \mathscr{H}_i, {}^{\text{in}}\mathscr{V}_i, {}^{\text{out}}\mathscr{V}_i, {}^{\text{in}}\partial_i, {}^{\text{out}}\partial_i, {}^{\text{in}}\mathscr{E}_i, {}^{\text{out}}\mathscr{E}_i, \mathscr{E}_i))_{i \in I}$
- a set of edges ${}^{\text{in}}\mathscr{E}$ - ie. a binary relation ${}^{\text{in}}\mathscr{E} \in \wp(\mathscr{V} \times (\bigcup_i {}^{\text{in}}\mathscr{V}_i))$ - each edge connects a vertex $v \in \mathscr{V}$ to an entry vertex ${}^{\text{in}}v_n \in {}^{\text{in}}\mathscr{V}_n$ of the sub-fractal hypergraph h_n,
- a set of edges ${}^{\text{out}}\mathscr{E}$ - ie. a binary relation ${}^{\text{out}}\mathscr{E} \in \wp((\bigcup_i {}^{\text{out}}\mathscr{V}_i) \times \mathscr{V})$ - each edge connects an exit vertex ${}^{\text{out}}v_n \in {}^{\text{out}}\mathscr{V}_n$ of the sub-fractal hypergraph h_n to an vertex $v \in \mathscr{V}$,
- a set of edges $\mathscr{E}$ - ie. a binary relation $\mathscr{E} \in \wp(\mathscr{V} \times \mathscr{V})$ - each edge connects a vertex v_o to another one v_d

Definition 3 (Fractal hypergraph) A fractal hypergraph is defined as:

- basic fractal hypergraph $\mathscr{HB} = (\mathscr{V}, \mathscr{H}, {}^{\text{in}}\mathscr{E}, {}^{\text{out}}\mathscr{E}, \mathscr{E})$
- a set of entry vertices ${}^{\text{in}}\mathscr{V}$, a set of exit vertices ${}^{\text{out}}\mathscr{V}$,
- a set of entry edges ${}^{\text{in}}\partial$ – ie. a binary relation ${}^{\text{in}}\partial \in \wp({}^{\text{in}}\mathscr{V} \times \mathscr{V})$ that connects the entry vertices ${}^{\text{in}}\mathscr{V}$ to vertices of the basic fractal hypergraph $\mathscr{V}$
- a set of entry edges ${}^{\text{out}}\partial$ – ie. a binary relation ${}^{\text{out}}\partial \in \wp({}^{\text{out}}\mathscr{V} \times \mathscr{V})$ connecting the entry vertices ${}^{\text{in}}\mathscr{V}$ to vertices of the basic fractal hypergraph $\mathscr{V}$

Fractal hypergraphs are a kind of powerful extension of hierarchical graphs[17] and can represent program code, hardware design or logical formulae.

Since *fractal hypergraphs* are hypergraphs where the hyperedges between the vertices are defined by hypergraphs, a fractal hypergraph can easily represent a complex system that mixes program code, hardware design and logical formulae, each hyperedge of the hypergraph may be either a fractal hypergraph that

represent a sequence of instructions (program representation), a sequence of logical operations (hardware representation), a logical formulae (specifications) or a complex subsystem that mixes the previous notions.

3.2.2 2nd step: reducing the size of memory

Standard approach for abstract debugging associates for each instruction point the values stored in the environment, which makes the memory size required explode, and it will result in developing a tool that can only be used to formally execute debug small sequence of codes. To avoid it, we decided to associate for each value the "hypervertice" where the value is first valid and the first one where this value isn't.

4 A small example

In this section, we will quickly explain how the formal debugger works and what kind of results we can expect. We consider the following SystemC[1] implementation of a small counter.

```
SC_MODULE(counter) {
  sc_in<bool> clock, load, clear;
  sc_in<sc_int<8> > din;
  sc_out<sc_int<8> > dout;
  sc_int<8> countval;

  SC_CTOR(counter) {
   SC_METHOD(onetwothree);
   sensitive_pos(clock);
  }
  void onetwothree() {
   if(clear)
     countval = 0;
   else if(load)
     countval = din.read();
   else  countval ++;
   dout = countval;
   }
};
```

The code first introduces the input signals: `clock` is the internal clock, `load` indicates that the counter should load the value from `din`, `clear` indicates that the counter should restart to count from 0. The output signal `dout` exports the counter value. The internal register `countval` contains the internal value of the counter. The constructor `SC_CTOR(counter)` tells that the method `onetwothree` is triggered when the clock goes from 0 to 1. Finally the method `onetwothree` implements the logical operations that updates the internal of the counter and sets the output value `dout`.

As as designer, we'd like to verify that the method `onetwothree` does the right job. Thus, we'd like to extract the logical formulae associated to the method.

[1] SystemC is a set of classes and constructions that allow to modelize hardware systems in C++

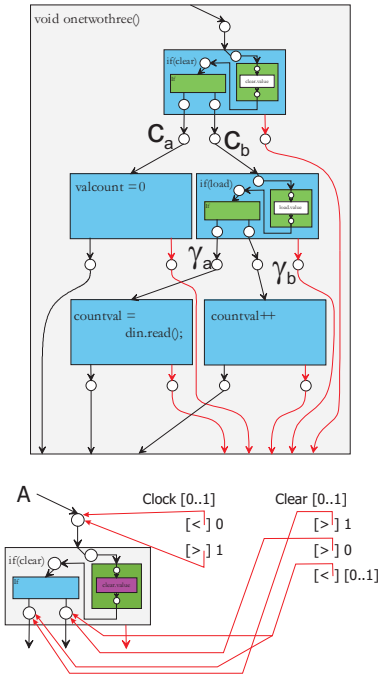

Figure 1: Hypergraphs and formal execution of the method *onetwothree*

The system first compiles the method into the fractal hypergraph presented in figure 1. Once it has been generated from the code, the symbolic execution on the fractal hypergraph can be started.

The second part of figure 1 illustrates the values inferred during symbolic execution of the first condition. For instance, we can see that the `clock` signal is 0 before the `onetwothree` method is triggered, 1 after the `onetwothree` method is triggered.

Depending on the result of the comparison, the symbolic execution deduces the various values that the signal `clear` has with respect to the resulting branch of the test. In the same way, the symbolic execution deduces the various values for the signal `load` has with respect to the resulting branch of the test.

The next step consists in adding counters to the hyperedges. A counter counts the number of time that an hyperedge has been activated. This approach has been introduced in abstract interpretation the number of times a loop is executed. Once the symbolic execution terminated, the formal debugger introduces counters that count the number of time a vertice has been activated. We concentrate for our example to the vertices that are active after the test. We call (c_a, c_b) the counters that are associated to vertices that are active at the end of the second test `if(clear)` and (γ_a, γ_b) the counters that are associated to vertices that are active at the end of the first test `if(load)`.

Once the abstract debugger has introduced the counters, we can now infer using standard abstract interpretation techniques [15] [14] the following prop-

erties when exiting the method:

- $c_a \in [0..1]$, $c_b \in [0..1]$
- $\gamma_a \in [0..1]$, $\gamma_b \in [0..1]$
- $c_a = \texttt{clear}$, $c_a + c_b = 1$, $\gamma_a + \gamma_b = c_b$
- `din.value` = v
- `valcount` = v_a when entering the method
  $(v_a + 1)c_b + v\gamma_a$ when exiting the method.

For the formal debugging of this simple example, we haved used: (1) symbolic execution and (2) abstract interpretation to infer value ranges[15], relational equalities [14] and execution counters [16].

5 Ongoing works

Our approach allows to define platforms with many communicating tools. The tool under implementation is based on an automatic analyzer of hardware components written in SystemC at many modeling levels and an automatic assembler of analyzer results through a project manager and a formal debugger.

As you can imagine, the automatic verification of a system or a component whose all components and system behaviors are not specified generates many warnings and potential errors. To debug the components, the user first connects additional "contract components" to send generic stimuli that adheres to the protocol supported by the component. After connecting the "contract components" to the system, the debugger generates the hypergraph that implements the protocol introduced by the additional "contract components". Then the user should look at each residual errors or potential violations to determine their origin: (1) if the error is a real hard one, it has to be corrected; (2) if the error is not an error per se but a state that should never occur, by clicking with the right button of the mouse on the code under each potential error, the user can choose first to exclude all the behaviors that may lead to this error, what automatically adds constraints that complements the previous "contract components" and refines the protocols supported by the current analysed system.
To decide what to do with each error, the formal debugger should assist the user to determine its cause. First, he can browse the trace simulation by simple clicking on the source simulation code and by on need instantiation of some signals or registers or variables. On each point of such a trace, the user can emit requests like "what is the value of that variable at this point expressed in constant, interval or hypergraph", "from what signals depends the value of that variable/register at this point". This interactive approach has been successfully tested for small hardware components and allows to find some errors that could not have been found with other standard verification techniques, inclusive model checking.

6 Future works

Along the ongoing implementation of the platform, we now investigate how to generate, starting from the specifications that get verified and extracted during the analysis of the transactional model of system architecture, a set of tests that the RTL level components of the architecture must pass to ensure that they correctly implements the system level architecture.

References:

[1] SystemC[online]. Available: http://www.systemc.org

[2] SpecC[online]. Available: http://www.spec.org

[3] F. Hague, J. Michelson, K. Khan. The art of verification with Vera. *Verification central, 2001*

[4] S. Palnitkar, Design Verification with e, *Prentice Hall, 2003*

[5] C. Norris Ip, S. Swan A Tutorial Introduction on the New SystemC Verification Standard, *in Proceedings of DATE03, IEEE Computer Society, March 2003*

[6] P. Chauhan, E.M. Clarke, Y. Lu, D. Wang, Verifying IP-core based on system-on-chip design. *In proceedings of ASIC conference, IEEE 1999*

[7] B. Bentley, Validating the intel Pentium 4 microprocessor. *In Proceedings of DAC 2001, ACM 2001*

[8] RNTL[online]. Available : www.telecom.gouv.fr/rntl/

[9] M.Behm, J. Ludden, Y.Lichtenstein, M. Rimon, and M. Vinov, Industrial Experience with Test Generation Languages for Processor Verification. *In Proceedings of DAC 2004. ACM, San Diego, California, June 7-11 2004, 36-40*

[10] L. Cai, D. Gajski *Transaction Level Modeling: an overview* In Proceedings of the CODES+ISSS03, ACM Newport Beach, October 1-3, 2003 ACM, 2003.

[11] T. Schubert High level formal verification of next-generation microprocessors. *In Proceedings of DAC 2003. ACM, June 2-6 2003.*

[12] R. Kaivola, N. Narasimhan. Formal Verification of the Pentium 4 Floating-Point Multiplier In Proceedings of DATE 2002. IEEE Computer Society, March 4-8 2002, 20

[13] M. Kaufmann, P. Manolios, and J. Strother Moore, Computer-Aided Reasoning: An Approach, *Kluwer Academic Publishers, June, 2000. (ISBN 0-7923-7744-3)*

[14] M. Karr. Affine relationships among variables of a program *Acta Informatica, 6:133–151, 1976*

[15] P. Cousot, N. Halbwachs Automatic discovery of linear restraints among variables of a program *In : Conference Record of the Fifth Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, Tucson, Arizona, 1978. pp. 84-97. - ACM Press, New York, New York, United States*

[16] A. Venet. Automatic Determination of Communication Topologies in Mobile Systems. *In : Proceedings of the Fifth International Symposium on Static Analysis, SAS '98, edited by G. Levi, pp. 152-167. - Springer, Berlin, Germany, 1998, Pisa, Italy, 14-16 september 1998, Lecture Notes in Computer Science 1503.*

[17] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *in Proc. Foundations of Software Science and Computation Structures (FOSSACS 2000), Lecture Notes in Computer Science, Vol. 1784, pp. 98113, Springer-Verlag, New York/Berlin, 2000.*

[18] F. Bourdoncle. Abstract debugging of higher-order imperative languages. *In : PLDI '93, Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, Albuquerque, New Mexico, 1993. pp. 46–55.*