Virtual Fault Simulation of Distributed IP-based Designs

Marcello Dalpasso[DEI], Alessandro Bogliolo[DI], Luca Benini[DEIS] and Michele Favalli[DI]

DEI – University of Padova – Via Gradenigo, 6A – 35131 Padova, Italy
DI – University of Ferrara – Via Saragat, 1 – 44100 Ferrara, Italy
DEIS – University of Bologna – Viale Risorgimento, 2 – 40136 Bologna, Italy

http://www.javacad.eu.org

Abstract

Fault simulation and testability analysis are major concerns in design flows employing intellectual-property (IP) protected virtual components. In this paper we propose a paradigm for the fault simulation of IP-based designs that enables testability analysis without requiring IP disclosure, implemented within the JavaCAD framework for distributed design [1, 2]. As a proof of concept, stuck-at fault simulation has been performed for combinational circuits containing virtual components.

1. Introduction

The need for reducing the time-to-market and design cost of today's integrated circuits pushes electronic designers to (re-)use third-party components. Re-usable components are commonly called *virtual components* or *intellectual-property* (IP) *components*, since they represent third-party's intellectual properties rather than hardware. IP-based design raises the issue of information transfer between the developers of virtual components (hereafter called *IP providers*) and the *designers* that may want to use them. Any information made available before IP disclosure should protect the providers, while containing enough data to allow the designers to evaluate IP components within their designs.

According to the recommendation of the Virtual Socket Interface Alliance (VSIA) [3], the *open* specification of an IP component should contain an executable functional model, pre-characterized static estimates of the main cost metrics, and testing guidelines and patterns. However, the need for protecting the provider's IP limit the accuracy of any open, instance-independent specification. In this context, testability analysis is one of the most challenging issues because: *i)* the testability of a component depends both on the controllability of its inputs and on the observability of its outputs within the design, and *ii)* there are no general exact composition rules for estimating the testability of a design from the testability of its components. Mainly for these reasons, the VSIA testability directives are much more complex than those regarding any other cost metric. In particular, they recommend the use of DFT techniques or BIST architectures to preserve the testability properties of each component, with such non-negligible costs in terms of performance and area, that are not always compatible with design budgets.

On the other hand, several approaches have been recently proposed to address testing issues at high levels of abstraction [4, 5, 6, 7]. Research in this field is prompted by different motivations: *i)* high-level testing can be used to verify by simulation the functional correctness of a design before performing low-level synthesis; *ii)* raising the abstraction level reduces the complexity of all design tasks; *iii)* high-level models can be used to steer design choices towards more testable solutions. All proposed approaches start by defining high-level fault/error models that do not require any knowledge about the inner structure of the components. However, the issue raised by IP protection is slightly different in nature: it is not about the verification of design correctness before low-level synthesis, rather, it is about the evaluation of testability properties of a design with IP components whose implementation cannot be accessed by the designer. Using high-level fault models in such a context would impair accuracy by disregarding valuable low-level information that is already available to the IP provider, even if not accessible by the designer.

In this work we propose an innovative solution (called *virtual fault simulation*) that allows the designer to perform accurate fault simulation of the entire design without disclosure of the IP-protected components. Fault simulation is performed by the designer, but whenever some knowledge about the implementation of an IP component is required, the provider is asked to perform such simulation subtasks. Designers and providers may be remotely located and they may communicate across the Internet. Internet connectivity is gaining increasing importance in electronic design automation [8, 9, 10, 11, 12, 13]: it offers new opportunities for remote tool integration, it enables new ways of information sharing and collaborative design, and it enhances the market penetration of IP components. The JavaCAD design framework has been developed to provide support for distributed IP-based design [1, 2], and it has been extended in this work to implement virtual fault simulation.

2. Virtual fault simulation

IP protection requirements raise a barrier at the boundaries of each virtual component: the designer cannot pass the barrier to look at the component implementation, and the provider cannot look outside the component when it is embedded in a design. Hence, the early estimation of any design metric before IP disclosure involves both providers and designers: providers have to supply reliable estimators for their components, while designers have to evaluate and compose the estimators associated with all components. At the cost of drastic approximations, typical component metrics (area, delay, power) can be treated as local/additive properties, and summed up to estimate the corresponding design metrics; providers can pre-characterize static estimators that do not contain any IP and distribute them.

Unfortunately, fault detection is not a local property. The detection of a fault of a component embedded in a larger design entails: *i)* signal propagation from the design primary inputs to the component inputs, *ii)* fault activation within the component, *iii)* fault propagation to the component outputs, and *iv)* error propagation from the component outputs to some design primary outputs. Steps *i)* and *iv)* involve the designer's IP, while steps *ii)* and *iii)* involve the provider's IP. Such an intertwined involvement makes it hard to precharacterize detectability metrics: providers should supply complete information about the detection properties of an IP component, namely, the output pattern produced by the component corresponding to any possible input configuration and to any possible component fault. This is a huge amount of information whose worst-case extraction time and representation size grow exponentially with the number of inputs and linearly with the number of faults. On the other hand, only a small subset of such information is actually exploited by a designer.

The above observations suggest that dynamic (rather than static) information should be exchanged at run time between designers and providers to enable the fault simulation of IP-based designs. The need of run-time communication between designers and providers was addressed by JavaCAD [1] to enable *context-dependent dynamic estimation* of traditional cost metrics, with improvements upon the accuracy provided by context-independent static estimators. Within JavaCAD, boundary information (*e.g.*, input patterns/statistics and output loads) are automatically collected for each IP component during functional simulation of the entire design and sent to the providers to allow instance-dependent low-level estimation of the cost metrics of interest [2].

We extend this paradigm to fault simulation by addressing some additional issues: *i)* fault detection depends on the observability of the faulty component outputs from the design primary outputs, and *ii)* a good test sequence is an IP by itself and may need to be protected. From the above discussion we can specify the requirements for the fault simulation of IP-protected designs:

1. the implementation of the virtual components is a provider's IP that must be hidden to the designer;
2. the functionality and the structure of the design are IP of the designer that must be hidden to the provider;
3. fault simulation is driven by the designer;
4. fault simulation sub-tasks are services that the provider supplies (at some cost) to the designer;
5. the results of fault simulation belong to the designer;
6. the amount of information exchanged across the Internet should be minimized;
7. no IP information must be sent across the internet.

We conceived virtual fault simulation as a two-phase process. In the first phase, a target fault list is built for the whole circuit: it is a local, additive property that can be precharacterized for each component and made available to the designers. To guarantee IP protection, symbolic names are used to represent internal faults. The fault list for the entire design is built by the designer by composing the symbolic fault lists of all components.

In the second phase, fault simulation is performed. For each designer's test pattern, the design fault-free behavior is simulated and the signal configuration at the IP component inputs is made available to the provider, who returns the corresponding *detection table*. A detection table provides a partial representation of the component testability properties corresponding to a given input configuration: it tells the designer which output pattern would be produced by the component in response to the given input pattern, in presence of each possible internal fault. Each row of the table associates an erroneous output pattern with the list of (symbolic) faults that would cause that error. To inject and simulate fault f of component M, the designer injects at the outputs of M the output pattern associated with f in the detection table of M (if it exists) and propagates its effects to the design primary outputs by simulating the fault-free behavior of all other components. If an error occurs, fault f is detected together with all other faults associated with the same output pattern in the detection table of M, and they can be dropped from the fault list.

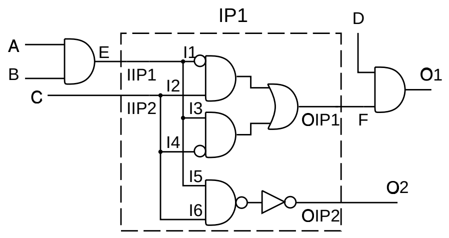

Figure 1. Example circuit with an IP block

Example 1 *Consider the simple circuit in Fig. 1, containing the IP block* IP1 *(a half-adder). The implementation of* IP1 *(hidden to the designer) is shown within the dashed box. The symbolic stuck-at fault list for* IP1 *is* {I1sa0, I1sa1, I2sa1, I3sa1, I3sa0, I4sa0, I5sa1, I5sa0, I6sa1} *(basic fault dominance has been exploited, while faults affecting input/output signals are directly handled by the designer).*

Suppose we want to evaluate whether or not input pattern ABCD=1100 *detects fault* I3sa0*. Propagating the input values* A *and* B*, we have* E=1*, hence the inputs of* IP1 *are* 10*. The detection table of* IP1 *corresponding to* IIP1=1, IIP2=0 *is:*

*Faulty Output (*OIP1, OIP2*)*	*Fault List*
11	{I6sa1}
00	{I3sa0, I4sa1}

Looking up the detection table for the faulty output values associated with I3sa0*, we obtain* 00 *(as opposed to the fault-free configuration* 10*). Unfortunately, such faulty value on* OIP1 *does not propagate to the primary output, because* D=0*; hence, pattern* 1100 *does not detect fault* I3sa0*. On the contrary,* I3sa0 *is detected by input pattern* 1101*, that gives rise to the same detection table (because the input configuration for* IP1 *is the same), but the faulty value at* OIP1 *now propagates to* O1*. Fault* I4sa1 *is detected by pattern* 1101 *as well, since it causes the same error at the outputs of* IP1*.*

3. The implementation

3.1. JavaCAD basics

The JavaCAD backplane is a set of packages written in Java, called *JavaCAD Foundation Packages* (JFP). Components and designs (*i.e.*, hierarchical collections of interconnected components) are sub-classes of the Module class. A Module object is specialized by overriding the methods that process events (*e.g.*, giving input/output functionality and evaluating cost metrics) and specifying connections. Modules are tied together by Connectors, that perform no other function but zero-delay Token-passing between Modules. Wrappers are sets of interconnected Modules that enable the hierarchical specification of a design.

Modules in a design can be either local or remote. A local Module runs completely on the designer's Java virtual machine. Remote Modules are sent from an IP-provider to the designer through the Internet, but some methods of its run on the provider's JavaCAD server.

The implementation of remote Modules is based on Java Remote Method Invocation (RMI), a CORBA-like protocol that makes distributed objects easy to implement and use. The key features of RMI exploited by JavaCAD are: *i)* creation of local instances of remote classes without having their byte-code available (such classes are IP Modules), *ii)* invocation of methods of remote classes, with a proper handling of parameters and return value, and *iii)* handling of secure client-server transactions.

IP protection is implemented by splitting remote Modules: the IP-protected part of the component specification is located on the provider's server as a private class, whose byte-code is never sent to the client, and the public part (an RMI stub) is freely distributed to the designers and is used to instantiate the remote component within a design. Remote method invocations are transparently handled by the stub. Finally, protection of the IP of the designer that uses remote components is guaranteed by bounding each Module with Connectors, that enables JavaCAD to completely inhibit the transmission of sensitive design information.

The JavaCAD event-driven simulation engine supports multiple event types (Tokens), that are scheduled and delivered by a Scheduler. Multiple Schedulers can be run in concurrent threads, thereby providing full support for concurrent simulations running over the same design. Notice that Tokens are not used to represent functional events only (*i.e.*, changes of signal values), but they provide a general communication paradigm to traverse the design, collect information from modules, set up run-time parameters, etc.

3.2. Estimation with JavaCAD

JavaCAD can perform static and dynamic estimation of any design metric, called Parameter. The evaluation of the actual value of a Parameter for a Module is performed by an Estimator, that can be either local or remote. This feature enables *virtual simulation*, *i.e.*, the early evaluation of a design metric that requires the knowledge of not-yet-disclosed implementation details.

Static estimation of context-independent parameters consists of four main steps: *i)* a StaticEstimationController runs its Scheduler sending an EstimationToken to the Design; *ii)* the EstimationToken handler of the Design dispatches the event to each Module, and this happens again every time a Wrapper handles the event, thus traversing the entire design hierarchy; *iii)* the EstimationToken event handler of each leaf Module fills the proper field of the EstimationToken with its own estimation of the required static parameter, that is the ParamValue returned by the selected Estimator (each StaticEstimationController can define its own CustomParamValue); *iv)* any Module returns its filled EstimationToken to the StaticEstimationController, that *merges* the estimated parameters using a function provided by the CustomParamValue.

Dynamic estimation of context-dependent parameters is similar to the static one, with the following main differences: *i)* a DynamicEstimationController does not run its own Scheduler, but works within the SimulationController that drives the functional simulation of the Design; *ii)* an EstimationToken is sent to the Design (and dispatched as previously described) at the end of each simulation time instant, when the event-driven simulation is in a steady condition; *iii)* the outcome of dynamic estimation is not a simple estimation value for the whole circuit, but a time-stamped trace of values (*history*).

3.3. Virtual fault simulation with JavaCAD

For the sake of simplicity, we restricted the scope of this preliminary implementation of virtual fault simulation to stuck-at faults in combinational circuits. The implementation of fault simulation in the JavaCAD architecture required the extension of the static and dynamic estimation capabilities of the original JFP. Static estimation is used to build the fault list (FL) of the design as the union of the symbolic fault lists of its components.

Dynamic estimation is used to actually perform the fault simulation. First, a DetectionTable class has been derived from ParamValue to represent the detection table introduced in Section 2. For a component M and a given input pattern, the DetectionTable contains the list of all possible erroneous output configurations of M and, for each error, the (symbolic) list of the internal faults that may provoke it (see Example 1). At the end of each simulation time instant each Module gets an EstimationToken to be filled with its DetectionTable. Current fault-free input signals are made available to the Module estimator to construct the corresponding DetectionTable. After collecting the detection tables for all design components, the DynamicEstimationController uses them to determine the actual faults detected by the current test pattern:

1. for each (not yet detected) fault f of Module M in FL, find the faulty output signal configuration s associated with f in the DetectionTable of M;
2. if s exists:
   (a) inject s in the fault-free design and simulate its effects, retaining the current signal values at the design primary inputs (as detailed later on);
   (b) if the primary output configuration is different from the fault-free one, each fault associated with s in the DetectionTable of M is detected.

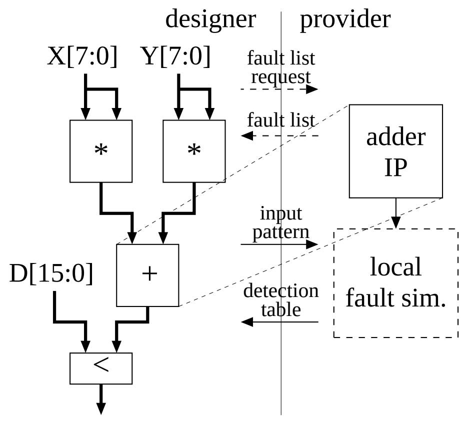

Figure 2. Case study

Detected faults are dropped from FL and annotated in the simulation *history* to represent the incremental fault coverage obtained with the actual test sequence.

To perform step 2a, a new SimulationController is instantiated with the following properties: *i*) it has only one simulation time instant; *ii*) primary inputs assume the signal values they have in the fault-free circuit at the current time; *iii*) the event-handling function of the faulty module M is replaced (for this Scheduler only) by a function that assigns the faulty signal configuration s to the outputs of M, independently of the input values of M.

This procedure is made possible by the JavaCAD event scheduling and handling, that has been conceived with concurrent multi-scheduler capabilities in mind. Hence, any event-handling in Modules is dependent on the actual Scheduler that is dispatching the event, and it has no effect on what performed by other schedulers. In such a way, we need no "reset" or save/restore action between different scheduler runs.

4. Performance analysis

Performance analysis and optimization of virtual fault simulation should be re-targeted as compared to classical fault simulation. In fact, the main bottleneck of virtual fault simulation is the communication between the designer and the provider over the Internet. Performance can be enhanced by minimizing both the amount of transferred information and the number of transactions.

We evaluate and discuss performance by referring to the case study of Fig. 2. The design evaluates whether or not the sum of squares of two 8-bit integer numbers (X and Y) exceeds a given threshold (D). The 16-bit sum is performed by a virtual component with protected IP, while in-house macros are used to implement the rest.

For IP components, both static and dynamic EstimationTokens are processed by remote methods that run on the provider's machine in order to access the implementation

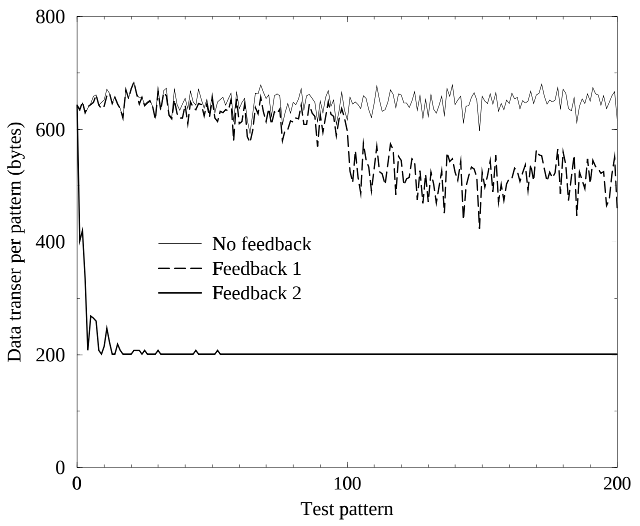

Figure 3. Data transferred across the Internet

of the component without violating the intellectual property. The invocation of a remote method entails a client-server transaction that requires a bidirectional data transfer: the client sends data to the server as parameters of the remote method, and the server sends data back to the client as return values of the remote method.

We are interested in measuring the amount of information exchanged across the Internet during virtual fault simulation of our case study. Curve labeled "No feedback" in Fig. 3 shows (for each test pattern) the communication cost of the virtual fault simulation of the design. The number of exchanged bytes depends on the size of the detection table, that depends on the current test pattern. The entire simulation run of 400 test patterns requires to transfer about 260 Kbytes (the amount of information that would be required to represent a priori all detectability properties of a 32-input macro would be thousands of Gbytes).

To reduce communication, we observe that the designer doesn't need any additional information about already-detected faults. If the designer provides some feedback about detected faults, the provider can prune the fault list and simplify the detection table accordingly before sending it to the designer. We call this technique *server-side fault dropping*. Its effects on data transfer are shown in Fig 3: the dashed curve (Feedback 1) is referred to our case study, while the solid curve (Feedback 2) is referred to a design where the outputs of the adder are completely observable. In both cases, data transfers decrease as fault simulation proceeds. If the outputs are highly observable (Feedback 2), the average size of the detection table becomes almost null (the data transfer reducing to the baseline overhead of RMI transactions) after a few test patterns since most of the internal faults are detected soon. Server-side fault dropping reduces the total amount of data transferred during virtual fault simulation to 210 Kbyte for Feedback 1, and 81 Kbyte for Feedback 2. As a final remark, notice that the feedback required to perform server-side fault dropping actually exposes the results of fault simulation to the provider. This is a prize the designer has to pay to speed-up simulation.

5. Conclusions

In this work we have shown the feasibility of fault simulation with intellectual property protection in a distributed design environment. The main goal of protecting the intellectual property of both the IP provider and the IP user has been achieved, and a proof of concept has been given for stuck-at fault simulation at the gate level.

References

[1] M. Dalpasso, A. Bogliolo and L. Benini, "Specification and validation of distributed IP-based designs with JavaCAD," in *Proc. of Design Automation and Test in Europe*, pp. 684–688, 1999.

[2] M. Dalpasso, A. Bogliolo and L. Benini, "Virtual simulation of distributed IP-based designs," in *Proc. of Design Automation Conf.*, pp. 50–55, 1999.

[3] VSIA, *VSIA Architecture Document, version 1.0,* http://www.vsi.org/library/vsi-or.pdf, 1997.

[4] S. Ghosh and T. J. Chackaborty, "On Behavior Fault Modeling for Digital Designs," *J. of Electronic Testing: Theory and Appl.s*, no. 2, pp. 135–151, 1991.

[5] F. Ferrandi, F. Fummi, L. Gerli and D. Sciuto, "Symbolic Functional Vector Generation for VHDL Specifications," in *Proc. of Design Automation and Test in Europe*, pp. 442–446, 1999.

[6] T. Riesgo, Y. Torroja, E. De La Torre and J. Uceda, "Quality Estimation of Test Vectors and Functional Validation Procedures Based on Fault and Error Models," in *Proc. of Design Automation and Test in Europe*, pp. 955–956, 1998.

[7] I. Ghosh, A. Raghunathan and N. K. Jha, "A design-for-testability technique for register-transfer level circuits using control/data flow extraction," *IEEE Trans. on CAD*, vol. 17, no. 8, pp. 706–723, 1998.

[8] L. Benini, A. Bogliolo and G. De Micheli, "Distributed EDA tool integration: the PPP paradigm," in *Proc. of Int.l Conf. on Computer Design*, pp. 448–453, 1996.

[9] L. Geppert, "IC Design on the World Wide Web," *IEEE Spectrum*, June 1998.

[10] M. J. Silva and R. H. Katz, "The case for design using the World Wide Web," in *Proc. of Design Automation Conference*, pp. 579–585, 1995.

[11] D. Lidsky and J. Rabaey, "Early power exploration – a World Wide Web application," in *Proc. of Design Automation Conference*, pp. 27–32, 1996.

[12] H. Lavana, a. Khetawat, F. Brglez and K. Kozminski, "Executable workflows: a paradigm for collaborative design on the Internet," in *Proc. of Design Automation Conference*, pp. 553–558, 1997.

[13] M. Spiller and R. Newton, "EDA and the Network," in *Proc. of Int.l Conf. on CAD*, pp. 470–475, 1997.