FP7-ICT-2009-4-248613

DIAMOND

Diagnosis, Error Modelling and Correction for Reliable Systems Design

Instrument: Collaborative Project

Thematic Priority: Information and Communication Technologies

Requirements & Concept of the Diagnostic Model (Deliverable D1.1)

Due date of deliverable: May 31, 2010
Actual submission date: June 28, 2010

Start date of project: January 1, 2010 Duration: Three years

Organisation name of lead contractor for this deliverable: Universität Bremen

Revision 1.5

Project co-funded by the European Commission within the Seventh Framework Programme (2010-2012)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices

For information, contact Dr. Jaan Raik, e-mail: jaan@pld.ttu.ee.

This document is intended to fulfil the contractual obligations of the DIAMOND project concerning deliverable D1.1 described in contract number 248613.

Table of Revisions

Version	Date	Description and reason	Author	Affected sections
1.0	March 9, 2010	Initial structure	G. Fey	All sections
1.1	March 26, 2010	Contributions of academic partners	G. Fey, G. Hofferek, R. Könighofer, E. Larsson, J. Raik, O. Rokhlenko	2, 3, 4
1.2	April 20, 2010	Revised Structure, added descriptions	G. Fey	All
1.3	May 3, 2010	Added references	G. Fey, G. Hofferek, R. Könighofer, E. Larsson, J. Raik, O. Rokhlenko	2, 3, 4
1.4	May 12, 2010	Updated diagnostic model	A. Finder	5
1.5	May 17, 2010	Revised introduction	G. Fey	1

Authors, Beneficiary

Görschwin Fey, Universität Bremen
Alexander Finder, Universität Bremen
Georg Hofferek, Technische Universität Graz
Robert Könighofer, Technische Universität Graz
Erik Larsson, Linköping Universitet
Jaan Raik, Tallinn University of Technology
Oleg Rokhlenko, IBM

Executive Summary

This document presents an overview of the anticipated holistic diagnostic model to be used in DIAMOND. The diagnostic model can be considered the “backbone” of the DIAMOND infrastructure. DIAMOND considers different application domains – diagnosis and correction – at different levels in the design flow – transaction, implementation and post-silicon.

The aim of the diagnostic model is to integrate those different views as much as possible, to capture the individual requirements by the different applications, We devise use cases for the different abstraction levels considered within DIAMOND. For each abstraction level the use cases span all applications. From these use cases the requirements and the diagnostic model are derived.

List of Abbreviations

CDFG - Control and Data Flow Graph
DoW - Description of Work [6]
FP7 - European Union's 7^{th} Framework Programme
HDL - Hardware Description Language
HLDD - High-Level Decision Diagrams
IST - Information Society Technologies
PSL - Property Specification Language
RT level - Register Transfer level = RTL
SMT - Satisfiability Modulo Theories
URL - Uniform Resource Locator
VHDL - Very high speed integrated circuits Hardware Description Language

Table of Contents

Table of Revisions iii
Authors, Beneficiary iii
Executive Summary iii
List of Abbreviations iii
Table of Contents v
1 Introduction and Overview 1
2 Localization and Correction at the Transaction Level 3
2.1 Semi-Formal Flow 3
2.2 Flow based on Simulation 4
3 Localization and Correction at the Implementation Level 7
3.1 Generic Flow based on Formal and Semi-Formal Methods 7
3.2 Dedicated Correction Flow for Control Logic 8
4 Localization and Correction Post-Silicon and In-Situ 11
4.1 Flow based on Semi-Formal Analysis of Soft Errors 11
4.2 In-situ Diagnosis & Correction 12
5 Resulting Extensible Diagnostic Model 15
5.1 Fault Modeling 15
5.2 Internal Modeling Infrastructure 15
6 Summary 17
7 References 19

1 Introduction and Overview

The diagnostic model can be considered the “backbone” of the DIAMOND infrastructure. DIAMOND targets different application domains – diagnosis and correction – at different levels in the design flow – transaction, implementation and post-silicon. Figure 1 gives the general overview of DIAMOND. The diagnostic model is developed within WP 1. The other workpackages build upon this model.

On one hand, the types of faults and bugs to be handled depend on the specific step during the design flow. On the specification and implementation level, typically design bugs that lead to reproducible errors are considered and have to be removed. When considering environmental behavior, soft errors due to transient faults are more important.

On the other hand, the level within the design flow considered also determines the feedback required by a designer or verification engineer. Again, when considering an implementation bug in a description of a design, a location in the source code given in *Hardware Description Language* (HDL) is of interest and potentially a suggested correction for that location. When analyzing errors that occur post-silicon also a potential origin of this error is of interest. However, in this case such an origin must be located in a netlist and potentially the type of fault leading to the observed error should be indicated.

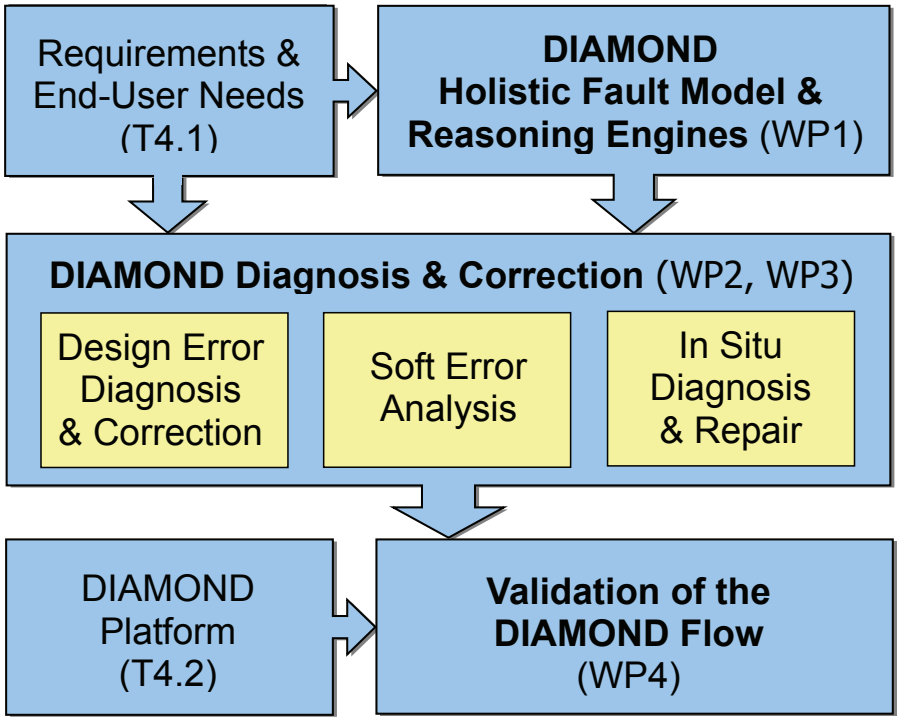

Figure 1: Overview of DIAMOND

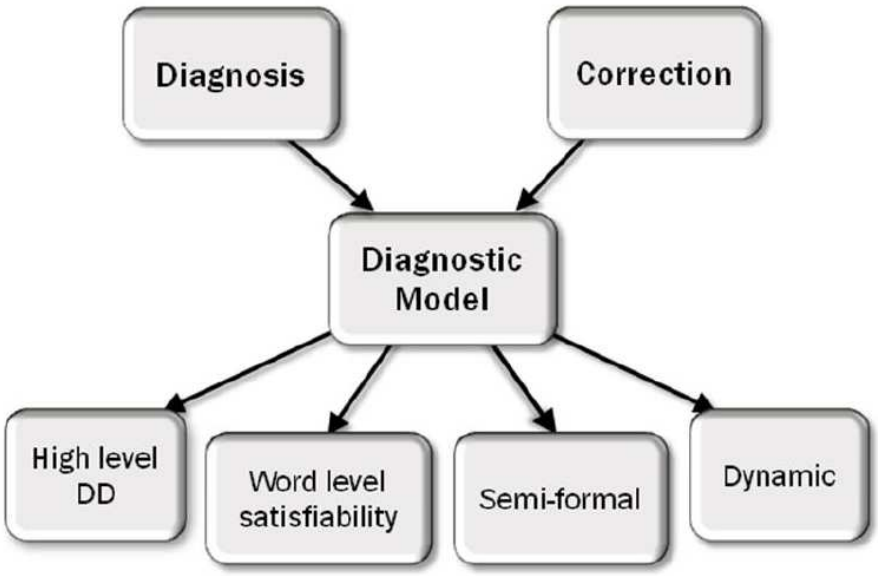

Figure 2: Diagnostic model within DIAMOND

The aim of the diagnostic model is to integrate those different views as much as possible, to capture the individual requirements by the different applications, but also to identify the distinguishing issues more clearly. Figure 2 taken from the *Description of Work* (DoW) [6] shows in more detail how the diagnostic model couples the application domains to the underlying formal, semi-formal, and simulation-based engines. In the following we devise use cases for the different abstraction levels considered within DIAMOND. For each abstraction level the use cases span all applications. These use cases obey to the requirements imposed by the end user requirements specified in Deliverable D4.1 [5] and they have the following form:

- *Input:* The design, any additional input required and the types of descriptions to be used.
- *Algorithm:* Brief description of the type of algorithms used and description of requirements on the diagnostic model from the algorithmic side.
- *Output:* The type of feedback expected from the algorithm and the infrastructure required to provide this feedback.
- *Requirements:* Per use case the requirements on the diagnostic model are explicitly given.

This document is structured as follows: Sections 2, 3, and 4 describe the use cases for diagnosis (WP 2 in [6]) and correction (WP 3 in [6]) at the transaction level, at the implementation level, and post-silicon/in-situ, respectively. With respect to the Dow [6] Section 2 is related to tasks T2.1 and T3.1, Section 3 is related to tasks T2.2 and T3.2, and Section 4 is related to tasks T2.3 and T3.3. Section 5 summarizes the requirements for the diagnostic model and indicates the concept of the model.

2 Localization and Correction at the Transaction Level

Design descriptions at the transaction level cannot easily be separated into meaningful smaller blocks. As a consequence formal methods alone may typically not be able to handle such descriptions due to resource limits. Therefore they will be extended by semi-formal and even purely simulation-based approaches. A simulation-based approach may be used as a fast preprocessing step for more accurate but also more time-consuming semi-formal approaches.

2.1 Semi-Formal Flow

Input: Specification and implementation.

The implementation is given as a piece of simple, imperative software (e.g., a simple C program). The term "simple" refers to the fact that not all constructs provided by the programming language will be supported. This includes concurrency features, floating point arithmetic, and to some extent also pointers.

The specification is not required to be complete in the sense that every execution of the program conforming to the specification is considered as a correct behavior. However, results are expected to be more accurate when using a more precise specification. The specification must allow to classify program executions into desired and undesired behaviors. Examples include specifications which are given as assertions in the program code, test vectors together with expected outputs, temporal specifications, or a golden reference model.

Algorithm: The algorithm performs symbolic [4, 13] or concolic execution [10, 16] on an abstract model of the program. For this purpose, the model must contain control flow as well as data flow information. It will therefore be similar to a *Control Data Flow Graph* (CDFG). The symbolic or concolic execution is used to explore possible execution paths of the program and to associate every such path with a path condition. The specification is used to classify execution paths into desired and undesired, thereby classifying the path conditions into two according sets. This way, reasoning about the correctness of the program can be reduced to the reasoning about the feasibility of path conditions in the two sets. This will in turn reduce to some sort of SMT solving. In order to actually perform diagnosis and correction, the algorithm identifies components in the abstract model of the program, which can serve as fault candidates. This is done according to the fault-model.

Output: In case of diagnosis a set of fault candidates is passed to the designer. A fault candidate is described by a certain portion of the source code. Correction extends each fault candidate by one or more suggested repairs. The repairs will be replacements of the fault candidates in the source code.

Requirements:

- Modeled faults are permanent, i.e. they do not change over time, as they correspond to bugs in the program.
- The relation to the source code must be known to allow for back-annotation of results.
- The abstract model of the program must contain control flow as well as data flow information.
- Decomposing the model of the implementation into components according to the fault model must be possible. These components serve as possible fault candidates.
- Replacing such a component with a different implementation must be possible.
- The fault model should be fine-grained, general, and allow for efficient algorithms.

2.2 Flow based on Simulation

Input: Specification, implementation and input stimuli.

Similar to the formal flow presented in the previous subsection, the implementation is given as a piece of simple, imperative software (e.g., a simple C program). Also the concept of the specification does not differ from the formal flow.

The semi-formal and simulation based flow adds the notion of input stimuli. In many practical cases (e.g. large designs, complex properties etc.) it is not feasible to solve error diagnosis and correction by formal means. In such cases it is possible to rely on existing test benches which the designer has created to verify the functionality of the system.

Algorithm: The algorithms are based on simulating the stimuli on the implementation. Similar to the formal approach a model based on a CDFG is used. Additionally, other representations for simulation derived from the CDFG model such as *High-Level Decision Diagrams* (HLDDs) [15], netlist representations, etc. may be implemented internally.

The error diagnosis is based on information obtained either by checking the assertions on the given simulation trace, or based on the mismatched outputs of the failing stimuli with respect to the output responses of the golden design. The fault models used in localization and repair are similar to the ones in formal methods.

Output: Output is similar to the one of the formal transaction-level methods. The most important difference is that semi-formal and simulation-based methods provide diagnosis and repair only with respect to the set of chosen input stimuli.

Requirements: The general requirements do not differ from the formal flow.

3 Localization and Correction at the Implementation Level

Different debugging problems will be focused in more detail and explained by use cases when considering the implementation level. This is due to the more comprehensive previous work on diagnosis and correction at the implementation level, e.g. [7, 9].

On the implementation level formal approaches will be considered as the core engines for diagnosis and correction. Using formal and semi-formal techniques on the implementation level is justified by smaller blocks that have to be handled. Semi-formal and simulation based techniques may be used for speeding up the overall process.

A generic approach will be applied to handle arbitrary debugging problems. This is complemented by a debugging approach that particularly focuses on faults in control paths.

3.1 Generic Flow based on Formal and Semi-Formal Methods

Input: Specification and implementation.

The implementation is given in terms of HDL source code. This source code can be transformed into a model suitable for formal methods by simple transformations.

The specification may be complete or partial. An example of a complete specification occurs in equivalence checking where the expected behavior is fully described by a reference design, a so-called *golden model*.

Partial specifications occur in model checking where a formal property describes a certain aspect of the behavior. Such formal properties may be given in a language similar to *Property Specification Language* (PSL) [1] or a subset of PSL.

Yet another type of partial specification are traces fully or partially specifying a starting state and values of primary inputs for the design that lead to the observation of erroneous output. The correct expected output response must also be specified in this case. A single or multiple traces may be given.

Algorithm: The algorithm runs on a formal model of the implementation to automatically perform localization and correction.

This model will be similar to a netlist of the design. In order to apply word-level reasoning engines like *Satisfiability Modulo Theories* (SMT) solvers, e.g. [19] or HLDDs [15], the model should extend beyond Boolean netlists also to more complex operations. Fault modeling and calculation of corrections are done at this level as well.

To achieve efficient localization and correction, techniques known to be efficient in formal verification will be applied. Namely, these are abstraction techniques, e.g. [3, 14], that allow to run the reasoning engines on smaller problem instances. Moreover, simulation may be used to further shrink the search space, e.g. in a pre-processing step [8].

If a golden model or a partial formal specification is available, this specification may be applied to find a set of qualitatively different counterexamples. Using such counterexamples improves the accuracy of the diagnosis step.

For correction two main strategies will be applied: a simple fault model that allows fast correction of a restricted set of bugs and a functionally complete but computationally expensive matching technique that allows to correct – in principle – any type of bug.

Output: In case of localization a set of fault candidates is passed to the designer. A fault candidate is described by a certain portion of the source code. Correction extends each fault candidate by a suggested repair. The repair will be a replacement of the fault candidate in the source code.

Requirements:

- Modeled faults are permanent, i.e. they do not change over time, as they correspond to bugs in the HDL.
- The instantiation hierarchy of the design must be known in the diagnostic model because identical instances will typically contain the same bugs.
- The relation to the source code must be known to allow for back-annotation of results.
- Fault model for a well-defined set of simple implementation level faults (bugs).
- Capability to hold a generic functional description for correction.

3.2 Dedicated Correction Flow for Control Logic

Input: A circuit on RT level and a functional specification.

The circuit is given either in an appropriate HDL or in form of a block diagram. The datapath of the circuit is complete, the control is either incomplete, (partially) incorrect, or missing at all.

The functional specification is given as a simple reference implementation. As an example, consider a simple, non-optimized implementation of a microprocessor, as

reference implementation for a highly optimized (and thus complicated) pipelined version of the same microprocessor. The reference implementation is also given either in an HDL or as a block diagram. It must make use of the same basic datapath blocks as the circuit under test.

Algorithm: From the circuit and the specification we extract an equivalence criterion, which is a formula in a fragment of first-order logic with arrays, uninterpreted functions, and equalities. We apply several transformations to this equivalence criterion to obtain an "equivalent" formula in propositional logic. From the latter we extract a repair for the incorrect control signals of the implementation, or new logic for missing control signals respectively.

Output: In case the original circuit contained faults in its control logic, the output will be a corrected version of the circuit. In case some or all control logic was missing, a completed circuit with newly synthesized logic for the missing parts will be outputted.

Requirements:

- The circuit under test and its reference implementation must make use of the same basic datapath blocks. These will be treated as uninterpreted functions by the algorithm. Examples for such blocks are Arithmetic Logic Units (ALUs) or Instruction Decoders of microprocessors.
- The circuit should contain some annotation which (control) signals are to be considered for correction/completion. This annotation can either be made manually or it can be the output of a diagnosis tool.

Related Work: The basic setting of this approach is similar to the verification approach applied by Burch and Dill [2] to pipelined microprocessors. Thus, the input will be similar as well. We will, however, extend the pure verification approach with the ability for synthesis/repair.

Srivastava et al. [17] also describe a way how a verification flow can be extended to be a synthesis/repair flow. Although their setting lies within the software world, the idea of generalizing verification to synthesis is similar. Suter et al. [18] also show how program fragments can be synthesized from pre- and post-conditions. However, their technique is more related to data manipulation then to control.

4 Localization and Correction Post-Silicon and In-Situ

To analyze the influence of soft errors and for post-silicon and in-situ approaches the full system has typically to be considered. This implies that mainly light-weight reasoning engines are used and that interfaces to actual hardware conform to existing standards. Both issues are discussed in the following use cases.

4.1 Flow based on Semi-Formal Analysis of Soft Errors

Input: Implementation and detection logic specification.

The implementation is given in terms of HDL source code or design pre-compiled to a gate-level.

The detection logic specification may be complete or partial. A complete specification describes an expected behavior of a detection logic in the referenced design. A partial specification can either describe a certain aspect of the behavior or provide only a list of detection logic end points, i.e. error flags.

Algorithm: A soft error, or a fault, is a transient bit-flip that occurs due to a particle strike. An error occurs when a fault results in data corruption. A fault does not always become an error; it may vanish through logical masking, electrical masking etc. In addition, fault-tolerant designs incorporate fault detection logic, which is capable of detecting that a fault occurred and initiating a correction or recovery sequence.

Whether or not a fault becomes an error depends on the state of the design when the fault occurs and on input values in subsequent cycles. The goal of soft error verification is to find faults that can result in errors without the error detection logic firing.

The algorithms run on the implementation to automatically perform the following tasks:

1. Vulnerability analysis, i.e. estimation of the probability of a fault to become an error for each sequential element.
2. Soft error rate reduction, i.e. applying local transformations on the design to reduce the overall vulnerability.

3. Detection logic coverage verification, i.e. validate that the fault detection logic covers all the sequential elements it should cover by the specification, plus a formal proof that the fault detection logic raises an error flag according to the specification.

Output: The output will consist of two reports:

1. A list of sequential elements ranked by their vulnerability to soft errors, before and after transformations.
2. A list of "holes" in the detection logic, i.e. a list of latches that are not covered by the detection logic for soft error faults, aside a proof of correctness of the fault detection logic in case it is up to the specification or a counterexample if it is not.

Requirements:

- Modeled faults are transient, i.e. they corrupt a single bit for one cycle.
- The instantiation hierarchy of the design must be known in the diagnostic model because identical instances will typically have the same soft errors vulnerability.

4.2 In-situ Diagnosis & Correction

Input: The specifications of software and hardware are given in a premanufactured format such as VHDL/HDL source and C source code. The hardware specification and the software specification are considered correct. Errors may originate from defectives in manufacturing, environmental impact, aging and errors that cannot be detected when analyzing software and hardware separately.

Algorithm/method:

- At post silicon diagnosis, the aim is to find timing errors that manifest themselves when software is executed on manufactured silicon. Hence, separate analysis of software and hardware will not find theses errors. An architecture compliant to P1687 [12] and the corresponding design optimization support will be developed.
- At in-situ diagnosis, the aim is to find errors, classify the errors and find the most appropriate way to repair the errors. The errors can be permanent (hard) or temporary (soft). A in-situ test architecture compliant to IEEE Standard 1149.1 [11] will be developed along with error classification scheme and selection of suitable repair.

Output: The output from post silicon part is an architecture and techniques to detect timing errors. The output from in-situ is an architecture and techniques to detect errors and suggest suitable repair for each error.

Requirements:

- Modeled faults are soft (transient) and hard (permanent).
- The fault model for in-situ should be in relation to possible repairs (replaceable units).
- A number of available repairs (for in-situ).

5 Resulting Extensible Diagnostic Model

This section provides a summary of the requirements from all use cases. These requirements directly define the capabilities expected from the holistic diagnostic model to be applied for different applications across different levels of the design flow from transaction-level over RTL to in-situ and post-silicon. While the project is running, the holistic diagnostic model may be extended and modified. In Deliverable 4.1 [5] user requirements are described that impose requirements on the use cases. Consequently, these are indirectly reflected in requirements for the diagnostic model.

The first subsection 5.1 describes requirements imposed by modeling bugs or hardware faults. The next subsection describes requirements on the infrastructure, i.e. on the front-ends to read the design description and on the internal representation.

5.1 Fault Modeling

The following list summarizes the requirements on fault modeling imposed by the previously described use cases.

- Faults are described at the logic level
- In the time domain permanent and transient faults are considered. For in-situ diagnosis besides a functional fault more fine grain timing issues have to be modeled.
- The portion of the design affected by a fault may be one or multiple components where a component may be any of the following
  - a line in the source code
  - a statement in the source code
  - a functional unit
  - a state element
  - a portion corresponding to a particular hardware structure

5.2 Internal Modeling Infrastructure

Requirements on the internal infrastructure of the diagnostic model derive from the expected input and output described in the use cases. In the following these requirements are listed:

- Representation of the design

  The representation of the design has to be efficient and easily integrable for different algorithms. The representation includes following features:

  - Links to source code,
  - Hierarchical information,
  - Differentiation between instantiation and definition,
  - Representation of control flow and data flow,
  - Decomposition into components.

- Specification

  The specification is a formal description of the expected behavior that covers different use cases. The specification must be represented in the diagnostic model and may be given in the following ways:

  - Reference design,
  - Partial formal specification,
  - Input stimuli and expected output responses.

- Diagnosis result

  Diagnosis is concerned with the identification whether the behaviour of a system is correct. For incorrect systems the hints to faulty parts of the design are given. This involves the following information:

  - Fault candidates,
  - Vulnerability information.

- Correction

  Faulty parts must be corrected which implies a subsequent change of the system that may be described as follows:

  - Replacement of components,
  - Generic functional corrections,
  - Simple transformations for corrections,
  - Description of available repair units.

In general, all requirements can be implemented in the internal representation. But *links to the source code* and *hierarchical information* also impose some necessities on the front-ends that read the design to fill the internal representation. Additionally, the internal representation structure may be filled by other user input, e.g. in case of specifying input stimuli that are not part of the design. Depending on the application and the back-end all of these aspects may be available or only selected ones.

6 Summary

In this deliverable the diagnostic model of the DIAMOND project is described. The aim of the diagnostic model is to integrate different views, like e.g. transaction level and implementation level with respect to different applications. Use cases for those views induce requirements on the infrastructure and on fault modeling. The integration of those requirements leads to a holistic diagnostic model.

7 References

[1] Accellera. *Accellera Property Specification Language Reference Manual, version 1.1*, 2004. last access 2010-04-26.

[2] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 68–80. Springer-Verlag, Berlin, 1994. LNCS 818.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[4] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[5] DIAMOND Consortium. D4.1 - definition of the diamond platform. DIAMOND - System Requirements & End User Needs, ICT 2009.3.2, 2010.

[6] DIAMOND Consortium. Description of work. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2010.

[7] M. Fahim Ali, S. Safarpour, A. Veneris, M.S. Abadir, and R. Drechsler. Post-verification debugging of hierarchical designs. In *Int'l Conf. on CAD*, pages 871–876, 2005.

[8] G. Fey, S. Safarpour, A. Veneris, and R. Drechsler. On the relation between simulation-based and SAT-based diagnosis. In *Design, Automation and Test in Europe*, pages 1139–1144, 2006.

[9] G. Fey, S. Staber, R. Bloem, and R. Drechsler. Automatic fault localization for property checking. *IEEE Trans. on CAD*, 27(6):1138–1149, 2008.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[11] IEEE. IEEE Std 1149.1-1990 – IEEE standard test access port and boundary-scan architecture-description. Technical report, IEEE, 1990. last access 2010-04-26.

[12] IEEE Working Group for IJTAG. IJTAG P1687. Technical report, IEEE, 2010. last access 2010-04-26.

[13] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[14] K. L. McMillan. Applications of Craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12, 2005.

[15] Jaan Raik and Raimund Ubar. Fast test pattern generation for sequential circuits using decision diagram representations. *Journal of Electronic Testing: Theory and Applications*, 16(3):213–226, June 2000.

[16] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[17] S. Srivastava, S. Gulwani, and J.S. Foster. From program verification to program synthesis. *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 45(1):313–326, 2010.

[18] P. Suter, R. Piskac, M. Mayer, and V. Kuncak. On Complete Functional Synthesis. Technical report, 2009.

[19] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. SWORD: A SAT like prover using word level information. In *VLSI-SoC: Advanced Topics on Systems on a Chip*, volume 291 of *IFIP Advances in Information and Communication Technology*, pages 175–192. Springer, 2009. Extended versions of best papers from VLSI-SoC 2007.