FP7-ICT-2009-4-248613

DIAMOND

Diagnosis, Error Modelling and Correction for Reliable Systems Design

Instrument: Collaborative Project

Thematic Priority: Information and Communication Technologies

Report on Reasoning Engines and Dynamic Techniques (Deliverable D1.4)

Due date of deliverable: December 31, 2011
Actual submission date: December 31, 2011

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

Organisation name of lead contractor for this deliverable: University of Bremen

Revision 1.9

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 D1.4 described in contract number 248613.

Table of Revisions

Version	Date	Description and reason	Author	Affected sections
1.0	October 4, 2011	Initial structure	A. Finder, G. Fey	All sections
1.1	October 13, 2011	TUG contribution	R. Könighofer	Model-based diagnosis, symbolic execution, concolic execution
1.2	October 18, 2011	UNIB contribution	G. Fey, A. Finder, R. Drechsler	Sections 2, 3, and 4
1.3	November 08, 2011	TUG contribution	G. Hofferek	Section 2.2
1.4	November 11, 2011	IBM contribution	E. Arbel	Sections 2.6, 2.7, 3.5 and 3.6
1.5	November 16, 2011	TUT contribution	J. Raik, U. Repinski	Sections 2.3, 3.3 and 4.1
1.6	November 29, 2011	TEDA contribution	S. Scholefield	Section 4.4
1.7	December 1, 2011	TL contribution	A. Jutman	Sections 2.8, 4.5, and 4.6
1.8	December 1, 2011	Revised all sections	G. Fey, A. Finder	All sections
1.9	December 5, 2011	Revised all sections	G. Hofferek, R. Könighofer	All sections

Authors, Beneficiary

Rolf Drechsler, University of Bremen
Görschwin Fey, University of Bremen
Alexander Finder, University of Bremen
Georg Hofferek, Graz University of Technology
Robert Könighofer, Graz University of Technology
Eli Arbel, IBM Research Labs in Haifa
Jaan Raik, Tallinn University of Technology
Urmas Repinski, Tallinn University of Technology
Artur Jutman, Testonica Lab
Stephen Scholefield, TransEDA Systems Ltd

Executive Summary

This document, which is an update to Deliverable D1.3 [4], presents an overview of the reasoning engines and dynamic techniques developed within DIAMOND. These formal, semi-formal, and dynamic reasoning engines are the core of the diagnosis, correction, and analysis approaches. Different types of engines are required, e.g., to trade reasoning power for resource requirements. Fine-tuning the reasoning engines for particular applications is necessary to achieve the required efficiency and the expected quality of results. This deliverable summarizes the work of Tasks T1.2 and T1.3 of work package WP1.

List of Abbreviations

ATPG - Automated Test Pattern Generation
BDD - Binary Decision Diagrams
CPU - Central Processing Unit
ESL - Executable Specification Language
FP7 - European Union's 7^{th} Framework Programme
FSM - Finite State Machine
HDL - Hardware Description Language
HLDD - High-Level Decision Diagrams
JTAG - Joint Test Action Group (a standard test access port - IEEE 1149.x)
LHS - Left-hand side
MBD - Model-Based Diagnosis
RHS - Right-hand side
RTL - Register-Transfer Level
SAT - Satisfiability
SMT - Satisfiability Modulo Theories
SoC - System on Chip
SUT - System Under Test
TAP - Test Access Port
TDI - Test Data In
UUT - Unit Under Test
WP - Work package

Table of Contents

1 Introduction and Overview

This deliverable is an update of Deliverable D1.3 [4]. To keep the contents self-explanatory a few parts of the general introduction are repeated in the following. Descriptions of activities refer to the previous deliverable where possible.

The approaches for diagnosis and correction that are developed within DIAMOND are implemented on top of different types of reasoning engines. Figure 1 gives a general overview of DIAMOND. Tools and concepts are developed within WP2 and WP3. Deliverables D2.2a [6], D2.3a [7], D3.2a [11], and D3.3a [12] describe tools and concepts that are currently under development. The diagnostic model, as defined in Deliverable D1.2 [10], connects the application level to the reasoning engines. Thus, the tools and concepts are implemented on top of the reasoning engines that are specified and created within WP1. Therefore the main objective is to provide reasoning capabilities as explained in the *Description of Work* [9]. Different types of reasoning engines are applied within DIAMOND:

- *formal engines* with strong reasoning capabilities, but potentially large resource requirements regarding run time and memory,
- *dynamic engines* with limited reasoning capabilities, but requirements in run time and memory are typically linear in the size of the input, and
- *semi-formal engines* that allow to shift from fully formal to purely dynamic engines and, by this, provide a trade-off between reasoning capabilities and resource requirements.

Different applications have varying requirements and lead to slightly different structures of the underlying problems. Therefore the reasoning engines can be tuned to a particular application and such tuning is necessary to achieve the required quality of results and acceptable efficiency of the computation. Moreover, the adaptation of certain types of reasoning engines is required to evaluate the trade-offs between the engines with respect to a particular application in the research project. Consequently, all parts of this reasoning infrastructure are developed to address the requirements stated in Deliverable D4.1 [8]. The main issues addressed on the level of reasoning engines are scalability, extended diagnosis support, and support for analyzing resilience of designs to soft errors.

A first status on the reasoning engines and dynamic techniques has been reported in D1.3 [4]. To be complete, also activities that already had the status *DONE* are listed in this document. However, to avoid repetition we refer to the preceding deliverable in corresponding sections.

This document is structured by the type of the reasoning engines in correspondence to the task structure of WP1, as outlined in the *Description of Work* [9]. As a consequence, the different types of reasoning engines used for a single application are reported in the respective subsections. The partners contributing to the work are briefly named in the header of each of the following sections. Subsection 1.1 gives

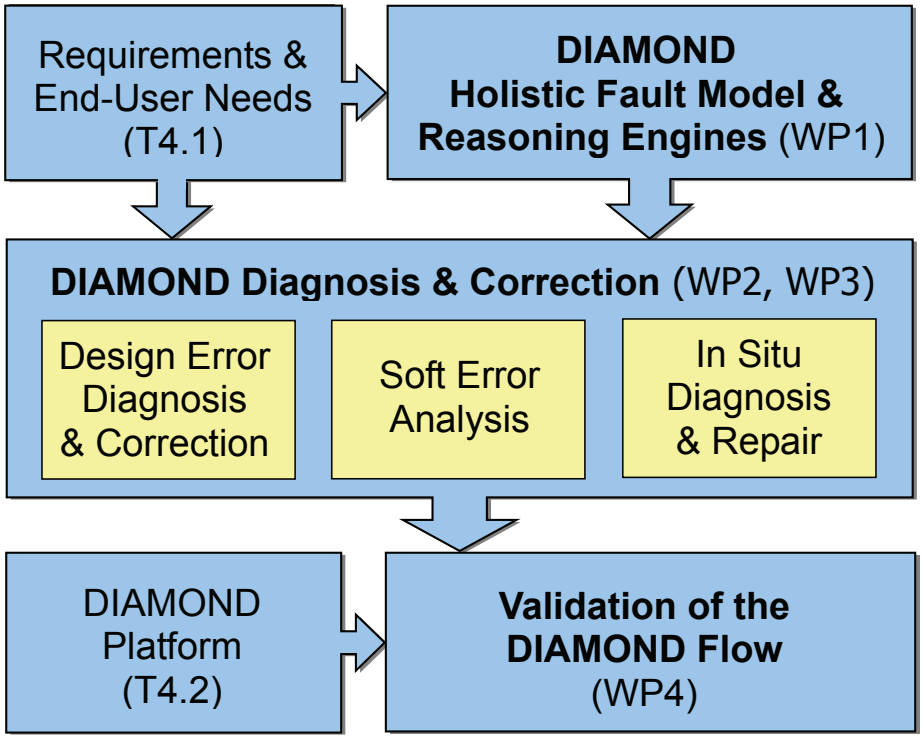

Figure 1: Overview of DIAMOND

a brief overview on cooperation between partners within WP1. The status labels for activities are explained in Subsection 1.2. Then, individual activities are reported in more detail. Section 2 reports the activities on formal reasoning engines under development, corresponding to the work performed within Task T1.2. The next two sections describe activities performed within Task T1.3, i.e., Section 3 reports the activities on semi-formal reasoning engines, and the progress on dynamic engines is reported in Section 4. Finally, a summary is given in Section 5.

1.1 Cooperations

TUT, TUG, and UNIB are jointly developing the tool FoREnSiC. FoREnSiC is a prototype tool to automate error localization and correction for system level specifications in C. A first open-source release is planned to the end of 2011. FoREnSiC is not only one engine but represents a set of several engines used to implement various debugging techniques. Furthermore, cooperation within WP1 has taken place between IBM and UNIB on the formal reasoning engines for the analysis of soft errors. The partners TUT and TL collaborated on reasoning engines for simulation with *High Level Decision Diagrams* (HLDDs).

1.2 Status Labels for Activities

For each activity the current status of the whole activity or of subtasks of the activity is reported by one of the following labels:

- *TO BE DONE* – detailed specification and implementation have not been started, yet
- *IN PROGRESS* – specification and implementation have been started, evaluation may have started already
- *DONE* – the activity has been completed, i.e., an evaluation of the approach has been done, academic partners have published results, industrial partners have decided how to use the results in their workflow

2 Formal Reasoning Engines

In this section the activities related to formal reasoning engines are reported. As explained in the introduction, these engines have been adopted to fit the needs of the applications considered within DIAMOND. The development of these engines was mainly driven by IBM, TL, TUG, TUT, and UNIB.

2.1 Activity: Model-based Diagnosis for Specification Debugging

This activity was already finished when Deliverable D1.3 *Status on the reasoning engines and dynamic techniques* [4] was submitted. A description can be found there.

Status of this activity: *DONE*

2.2 Activity: Certificate Extraction from Quantified Non-Boolean Formulas

As we have outlined in Section 2.2 of Deliverable D3.2a [11], we have developed a method to state repair problems as quantified non-Boolean formulas. The signals for which we want to find repairs are existentially quantified in these formulas. By computing certificates for these signals, we effectively compute valid repairs.

We have shown a preliminary proof-of-concept method to extract certificates from the relevant class of formulas [17]. This method was based on Binary Decision Diagrams (BDDs) and reduction of the problem to propositional logic. As this proved to be very inefficient, we have since then worked on improving the reasoning engine for certificate extraction.

We transform the problem into an interpolation problem, along the lines of [18]. Furthermore, we avoid the costly reduction to propositional logic by directly interpolating in first-order logic with equality and uninterpreted functions [20, 16]. Preliminary experiments show significant improvement over the BDD-based method. A more thorough practical investigation and an extension to multiple signals being repaired simultaneously is ongoing work within WP3.

Status of this activity:

- Certificate extraction by interpolation: *DONE*
- Extension to multiple signals to be repaired (theory): *DONE*

2.3 Activity: Diagnostic Test Pattern Generation

The HLDD-based sequential ATPG engine was already *DONE* in Deliverable D1.3 *Status on the reasoning engines and dynamic techniques* [4]. The description of this activity can be found there. The engine has been updated with the formal proof of untestable faults [24] and it will be applied to diagnostic test pattern generation for design error diagnosis in task T2.2 of WP2.

Status of this activity: *DONE*

2.4 Activity: Formal Latency Analysis

Latency analysis is a technique to support localization and correction of bugs and transient faults. For the computation of latency formal, semi-formal, and dynamic techniques have been applied. The status of these activities was *IN PROGRESS* when Deliverable D1.3 [4] was handed in. A detailed description about the contents of the activities can be found there.

Meanwhile, several extensions to this technique have been developed. First of all, a latency analysis of various circuits turned out that the *maximal latency* in many cases was hard to compute. Therefore, a loop detection technique has been added to determine whether the analyzed circuits contain sequential loops such that the maximal latency can be infinite. Furthermore, the analysis has been extended such that it can be applied selectively to parts of a circuit, e.g., memory elements or certain signals. This further supports a designer in understanding observed behavior. Additionally, a structural analysis has been performed which identifies shortest structural paths between two signals to find lower bounds and determines sequential loops efficiently. Results have been published in [15].

Status of this activity: *DONE*

2.5 Activity: Interfacing High-Level Descriptions with RT-Level Engines

The increasing complexity of circuit and system design is forcing design to move to higher programming languages above RTL, like ESL and C-based HDL. Thus, often a "golden model" is written in ANSI-C or a software-like language. The typical approach is then to convert this "golden model" into a hardware description language like VHDL or Verilog. Verifying equivalence, to determine if the C and HDL designs are consistent, is essential.

This activity is focussing on equivalence checking of timed HDL designs and untimed C programs. Both the C specification and the HDL implementation are translated into an intermediate representation. For this purpose, the C specification is mapped to a Finite State Machine (FSM) such that we are not restricted to unwind the specification to a bounded number of iterations of loops in the design. Moreover, the C specification is not restricted to be cycle accurate.

However, verifying equivalence between the high-level specification and the RTL implementation is an extremely hard problem. To reduce the size of the equivalence checking problem, we are looking for cutpoints. Cutpoints represent parts within both designs (the specification and the implementation) which are assumed to be equivalent such that the verification can be reduced using the cutpoints as starting points. Potential cutpoints are determined using simulation and verified by equivalence checking. On top of this reasoning engine for equivalence checking, diagnosis and repair algorithms will be created within WP2 and WP 3, respectively.

Status of this activity:

- Equivalence check between timed C and untimed HDL descriptions: *DONE*
- Determination of cutpoints: *DONE*

2.6 Activity: Latches Vulnerability Estimation

Latch vulnerability estimation is aimed to provide information regarding how a given latch is protected against soft errors. The use of formal analysis methods for this purpose allows getting a formal proof regarding the robustness of latches, rendering them fully protected and thus excludable from further checks, e.g. based on error injections. In case a formal proof regarding the robustness of a given latch cannot be obtained, either due to imperfect error protection or due to computational limitations, an approximation of latch vulnerability related to other latches can still be obtained using formal methods. As stated in D1.3 [4], this activity was divided into two corresponding parts: 1) proving latch robustness using formal methods

and 2) estimating latch vulnerability using efficient state space exploration techniques. The status of each sub-activity is described in the following.

- Finding robust latches and proving their robustness formally: A method based on formal verification was co-developed by IBM and the University of Bremen. The method works by using two copies of the design, where a fault model for single event upsets is introduced into one of the copies while the second one is left clean as a reference. A property construction asserting that under the same inputs and a single injection imposed by the fault model both models will produce the same outputs is also added to the model. Then a SAT-based model checker with interpolation is used for checking the property for each latch in the design. Latches for which the property passes are proven to be robust. Other latches are classified as non-robust if the injection becomes observable at output or unbounded robust if a soft error hit on them cause state corruption which is not observable on the outputs.
- Latch vulnerability using formal analysis: A model for estimating latch vulnerability using reachability analysis and 3-valued abstraction has been partially defined by IBM and the University of Bremen.

Status of this activity:

- Finding robust latches and proving their robustness formally: *DONE*
- Latch vulnerability using formal analysis (theory): *DONE*

2.7 Activity: Error Checker Coverage Analysis

A tool for identifying internal error checkers based on library conventions, as well as based on supplied user input was developed as part of the latch coverage analysis development (see Section 3.5). Once identified, internal error checkers are used for generating the model for the formal robustness analysis method mentioned in Section 2.6, specifically in the property construction.

In addition, an approach for decomposing the design into small pieces of logic, based on error checking windows, was developed. An error checking window is defined as the of piece logic (including latches) which is protected by a particular error checker. Once an error checker is identified and its error checking window is extracted, a formal engine can be used on the error checking window in order to check whether soft error hits on the latches inside the window cause the checker to fire. This approach is orthogonal to the type of error checking and allows proving latch coverage of various types of error checkers. A method for identifying error checking windows of parity-based checkers is being developed under WP2.

Status of this activity: *DONE*.

2.8 Activity: HLDD-Based Modeling Framework

Most parts of this activity have been implemented as and therefore shifted to "Semi-Formal Reasoning Engines". This is descirbed in Section 4.6. The results have been published in [22].

Status of this activity: *DONE*.

3 Semi-Formal Reasoning Engines

The status of the semi-formal reasoning engines within DIAMOND is summarized in this section. These engines were developed by IBM, TUG, TUT, and UNIB.

3.1 Activity: Symbolic Execution

Symbolic execution is a program analysis technique which allows to reason about program correctness under certain conditions. This information about the correctness of the program is used to perform automated error localization and correction on the transaction level. The basic idea and goals for a symbolic execution engine have already been introduced in Deliverable D1.3 [4]. To avoid repetition, this section discusses improvements over the status as reported in Deliverable D1.3 only.

The following sub-activities have been identified in Deliverable D1.3. Meanwhile, they are all done.

- Model suitable for symbolic execution: *DONE*
- Prototypical front-end for early experimentation: *DONE*
- More powerful front-end based on the `gcc` compiler and the `GIMPLE` intermediate language[1]: *DONE*
- Symbolic execution engine for simple programming constructs using the theory of linear integer arithmetic as an underlying reasoning theory: *DONE*
- Back-end which allows to apply implementation level diagnosis techniques on the system-level: *DONE*
- Extensions to support more complex programming constructs and theories: *DONE*

Apart from that, some optimizations have been performed. For the purpose of error localization and correction, not only the behavior of the original program has to be analyzed. It also has to be analyzed how the program would behave if certain components of the program could be replaced. In principle, this can be done in the following way. Let `LHS = RHS;` be an assignment with its left-hand

[1] `http://gcc.gnu.org/wiki/GIMPLE`

side (LHS) and right-hand side (RHS). Assume further that the RHS is considered as a component c which may be faulty. The assignment can be modified to

```
if(assume_correct(c)) LHS = RHS; else LHS = repair_c();
```

before performing symbolic execution. Here, `assume_correct` indicates whether the component c is assumed to be correct or faulty. `repair_c` is a yet unknown function which can return any value. The goal of error localization is to find out which components are assumed to be faulty, the goal of error correction is to find suitable implementations of the functions `repair_c` for the incorrect components. In order to enable this, both branches have to be analyzed. Depending on the number of components in the program, this can blow up the number of execution paths dramatically. Symbolic execution analyzes the program path-by-path, so it is crucial for the performance to avoid this blow-up.

A simple solution is to make the symbolic execution engine handle potentially faulty components of the program in a special way. The engine behaves as if component c always returns some unknown value r. Additionally, it stores the fact that $r = LHS$ if the component c is assumed to be correct.

The symbolic execution engine has been implemented from scratch as a part of the transaction-level diagnosis and repair tool FoREnSiC (see also Deliverable D2.1 [5]). Since FoREnSiC is going to be released as an open-source software, it will be shared with the community to stimulate further research in the fields of program analysis, error localization, and correction.

3.2 Activity: Concolic Execution

The concolic execution engine serves the same purpose as the symbolic execution engine: it is able to analyze a program for correctness under certain conditions, which is necessary to be able to perform automatic diagnosis and repair. The main concepts, the goals, and the differences to symbolic execution have already been presented in Deliverable D1.3 [4]. This section reports the improvements regarding the concolic execution engine with respect to Deliverable D1.3 [4] only.

The following sub-activities have been identified in Deliverable D1.3. Meanwhile, they are all done.

- Experiments with the concolic execution tool `CREST` [2], resulting in a prototypical extension that is able to check whether a given expression is a valid repair for a given faulty statement: *DONE*
- Instrumentation based on `CREST`, which in turn uses `CIL` [21]: *DONE*
- Concolic execution engine supporting simple programming constructs and the theory of linear integer arithmetic using `CREST`'s search strategies for execution paths; using (parts of) the symbolic execution engine for symbolic execution (cf. Section 3.1): *DONE*
- Extensions to support more complex programming constructs and theories: *DONE*

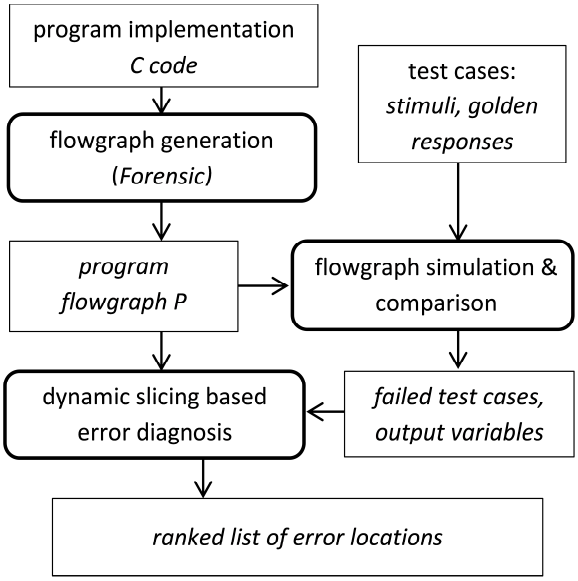

Figure 2: Dynamic slicing based error diagnosis flow

The performance optimization described in Section 3.1 has been performed for the concolic execution engine as well. This engine has not been developed totally from scratch, it is an extension of the concolic testing tool `CREST` [2], an open-source tool. `CREST` can only use the SMT-Solver Yices [13] with linear integer arithmetic, and it cannot handle potentially faulty components. It was extended to support also other SMT-Solvers such as Z3 [3] as well as SMT-Lib 2 [1] compliant solvers, bit-vector arithmetic, and to identify and analyze potentially faulty components. The concolic execution engine is part of FoREnSiC and will be shared with the community to trigger more research in the field.

3.3 Activity: Dynamic Slicing with Mutation-Based Repair

Figure 2 explains the error diagnosis process implemented in the dynamic slicing-based method. First, the FoREnSiC tool is applied to generate the flow graph representation of the program. All the test cases are simulated on the program flow graph and the obtained responses are compared to the golden output values of the tests which are provided for the diagnosis tool. Then, for the output variables receiving erroneous simulation values in the failed test cases, dynamic program slices are calculated. We refer to such slices as the failed slices. Each statement (i.e. node in the flow graph of the program) gets a score that shows the number of failed slices in which it was included. Statements are ranked according to the score. The ones with a higher score are considered to be more likely the root causes of errors in the design.

Table 1: List of mutation operators for repair

Mutation operators	C operators/examples
AOR (arithmetic operator replacement)	$+, -, *, /, \%$
ROR (relational operator replacement)	$==, !=, >, <, >=, <=$
LCR (logical connector replacement)	$\&\&, \|\|$
ASOR (assignment operator replacement)	$+=, -=, *=, /=, \%=, =$
UOR (unary operator replacement)	$+, -, \sim, !$
Bitwise operator replacement	$<<, >>, \&, \|, \hat{}$
Bitwise assignment operator replacement	$<<=, >>=, \&=, \|=, \hat{}=$
Increment/decrement operator replacement	$x++, ++x, x--, --x$
Number mutation (decimal digit replacement in integers, floats, and array indexes)	$0 \ldots 9$
Constant replacement (unary minus, unary plus, zero)	$+C, 0, -C$

After dynamic slicing, mutation-based repair of error candidates is performed. Mutation is a process, where syntactically correct functional changes are inserted into the program. Traditionally, mutations are performed by perturbing the behavior of the program in order to see if the test suite is able to detect the difference between the original program and the mutated versions. The effectiveness of the test suite is then measured by computing the percentage of detected, or killed, mutations. Here, we apply mutation operators for repairing erroneous circuits. The goal is to develop an error-matching-based repair approach, which would be capable of modeling realistic design errors. Moreover, it is crucial to select a limited number of mutation operators, because the perturbation and simulation of erroneous design implementations with a large number of error locations and mutant operators would become prohibitively time-consuming.

Table 1 presents the set of mutation operators which were implemented in the error-matching based repair method that we developed. The mutation operators include replacement of C language operators, which have been divided into several groups: arithmetic operators, relational operators, assignment operators, unary operators, etc. In addition, number mutations are performed by replacing each decimal digit in the numeric values one-by-one with other decimal values. This includes both, integer and floating point numbers and it covers also array indexes. Also, constants are mutated by inserting unary operators + and - as well as replacing the constants by zero.

Status of this activity:

- Dynamic slicing based diagnosis engine: *DONE*
- Mutation-based repair engine: *DONE*

3.4 Activity: Three-Valued Latency Analysis

In analogy to the formal latency analysis described in Section 2.4, the technique introduced in D1.3 [4] has been extended.

Status of this activity: *DONE*.

3.5 Activity: Latch Coverage of Error Detection Logic

A tool for performing latch coverage analysis of error detection logic has been implemented. Based on internal error checkers, the tool is able to list all latches which are not connected to any error checker. In addition, a specification language has been developed for enabling users to classify error checkers into several pre-defined types. This classification is used by the tool for generating valuable reliability-related statistics about the design. These include estimations on design exposure to silent data corruption events and other checkstopping events.

Status of the activity: *DONE*.

3.6 Activity: Latches Vulnerability Estimation

In addition to the formal analysis technique described in Section 2.6, semi-formal techniques for latch vulnerability estimation have also been developed in order to cope with large designs which the formal method cannot handle. At the core of the semi-formal method lies the notion of logical masking: The more logical masking a latch has, the less vulnerable it is to soft errors. Thus the purpose of the vulnerability method is to compute the logical masking of each latch in the design with respect to primary design outputs or designated state bits. This is accomplished by approximating error-propagation probabilities of each latch by means of performing observability analysis. A method based on computing latch-to-latch observability using BDDs was implemented. By building a latch dependency graph and annotating the graph edges with the computed probabilities, multi-cycle error propagation probabilities could be computed. This approach is suitable for moderately-sized designs, as the BDD-based computation sometimes is a bottleneck of the computation. A more robust approach was developed as well, based on the method published in [19]. This method computes local observability given input patterns generated from simulation. We have extended this method to take into account internal error checkers as well as designated state bits during the vulnerability estimation process. In addition, we extended the algorithm by allowing a user to specify simple inputs constraints (e.g. one-hot) so that the tool

can generate input patterns by itself. This allows users to perform vulnerability estimation early in the design cycle, regardless of input patterns availability.

Status of this activity: *DONE*.

4 Dynamic Techniques

Dynamic engines are a powerful approach to analyze very large systems. Additionally, dynamic engines are applied to further utilize results returned from other, more powerful engines. The dynamic engines and reasoning frameworks were designed and implemented by TEDA, TL, TUT and UNIB.

4.1 Activity: High-Level Decision Diagram Simulation Engine

This activity is finished and the details were already presented in Deliverable D1.3 *Status on the reasoning engines and dynamic techniques* [4].

Status of this activity: *DONE*

4.2 Activity: Evaluation of Debugging Algorithms

One way towards generalizing the results of debugging algorithms is the use of fault models to assess the performance of an algorithm for certain types of design bugs. In [14], an extensible fault model that describes different bugs in SystemC descriptions has been presented to evaluate debugging algorithms from a qualitative perspective.

Since simulation-based procedures are used in different areas of application, e.g., debugging, testing, compiling, a simulation-based algorithm has been developed in a first case study, to evaluate the presented fault model. By focussing the attention of a user on a subset of a design which is expected to contain faulty code, the objective of the simulation-based procedure is the reduction of the debugging effort. Thus, for a given design in SystemC, counterexamples are simulated to generate so-called *traces*. Each trace represents a subset of program statements (in other words: a subset of the design) and is expected to contain faulty code. Otherwise no counterexample should be generated. The intersection of all traces still includes and localizes the faulty part of the design. However, this assumption is restricted only to single bugs in a design. The principle of this algorithm is shown in Figure 3. The results of this activity have been presented in [14].

Status of this activity: *DONE*

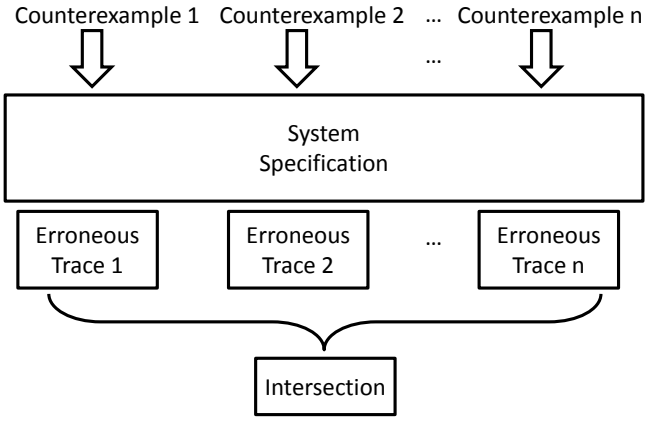

Figure 3: Simulation-based debugging

4.3 Activity: Simulation-Based Latency Analysis

In general, simulation-based procedures are able to efficiently handle large systems. The disadvantage of simulation is incompleteness in the sense that for large systems not all input sequences can be considered in reasonable time. In addition to the formal and semi-formal approaches (see Section 2.4 and Section 3.4), a simulation-based procedure has been developed and investigated for latency analysis. Compared to Deliverable D1.3 [4], and similar to the formal latency analysis, the simulation-based procedure has been extended with loop-detection and a selective application to parts of a circuit.

Status of this activity: *DONE*

4.4 Activity: Assertain Simulation, Hardware Accelerated Simulation, and Formal Engine

TransEDA has developed a sophisticated simulation, hardware-accelerated simulation, and formal engine flow in their product Assertain that are described in the following.

Assertain Simulation The core Assertain platform combines both dynamic simulation and a formal engine to provide a two-step diagnosis. The entire Assertain platform has been re-written to provide an *N*-step diagnosis so that partner tools can be added into the flow at different stages of the validation. The *N*-step diagnosis is built on a new work-flow engine so that any number of diagnostic steps

can be built up as an XML script and run by the work-flow engine in sequence. A specific diagnosis techniques can be targeted at specific code segment on Step One, another technique on Step Two and another on Step Three – and so on.

The Assertain platform has been re-written to accept XML inputs to initiate a tool set and report the output of the tool set to a central database. Most of the core modules have been re-coded and work is under way to integrate stand-alone tools by means of XML in WP2 and WP3.

Hardware Accelerated Simulation TransEDA has spent time integrating their Assertain simulation environment with the *Cadence Extreme Simulator*[2] with active clients in France and India. The technique inserts instrumentation into the hardware simulation environment and tracks the code coverage and other metrics performed by the standard Assertain dynamic engine - but at a very much faster speed. TEDA has successfully integrated the *Cadence Extreme Simulator* and work has been done on integrating the *Cadence Palladium Simulator*. A deeper analysis and evaluation of the simulator is ongoing work in WP2 and WP3.

Integration with a Formal Engine TransEDA uses the formal engine from AerieLogic[3] of France. This engines forms the second part of the two-step integration flow. The formal engine will be completely integrated into the *N*-step flow in WP2 and WP3.

Status of this activity:

- Assertain simulation for the *N*-step work-flow platform : *DONE*
- Hardware accelerated simulation : *DONE*
- Integration of formal engine (interface definition): *DONE*

4.5 Activity: Processor Centric Reasoning Framework

A detailed description of this activity was given in D1.3, when the status was *IN PROGRESS*. Now the activity is finalized and the results have been published in [23].

Status of this activity:

- Modeling workflow: *DONE*
- Synthesis workflow: *DONE*

[2] `http://www.cadence.com`
[3] `http://www.aerielogic.com`

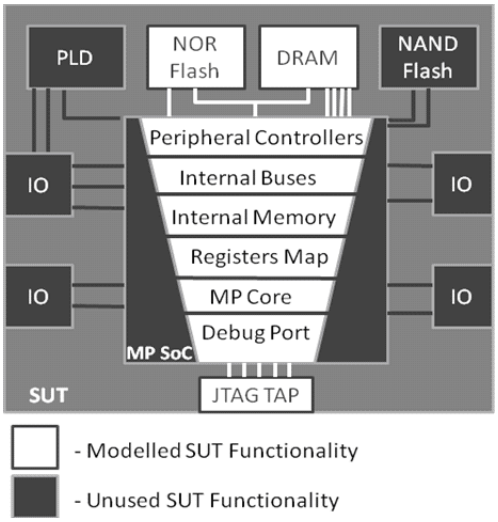

Figure 4: Test access path modeling and activation

4.6 Activity: HLDD-Based Diagnostic Access Modeling Concept

This activity is tightly connected with the activity in Section 4.5 described in detail in D1.3 [4]. The main goal is to develop a modeling framework that supports the reasoning framework from the previous activity. Tight cooperation with TUT enabled TL to utilize an efficient modeling framework – HLDD graphs – for automatic synthesis of diagnostic access path in *System on Chip* (SoC) based systems. The following text gives a short overview of the modeling methodology while details and experimental results are given in [22] and [23].

The test access routine is aimed at set-up of the communication between the external tester and the microprocessor in the SoC via the debug port (JTAG connector on the board). The second goal of the test access is to establish connection between the microprocessor and the *Unit Under Test* (UUT). All the test routines are typically fed trough the CPUs/SoCs debug port, either to the embedded memory or to the external system memory on the board. In both cases the access to the memories is provided by the means of the SoCs infrastructure including memory controllers and bus matrices. The full data path between the external test controller and the UUT includes the debug port, CPU core, firmware running on the CPU, peripheral UUT controllers, interconnection structure between the SoC, and the target UUT. Modeled components of the *System Under Test* (SUT) form a test path that begins at JTAG port of the SUT and ends in UUT as shown in Figure 4.

This long data propagation chain has to be properly handled for both test application and test access (propagation) purposes. In general, the test access requires tuning the respective controller in the SoC to communicate with the specific UUT. If the appropriate controller is missing, the test application routine should also handle the data transfer protocol of the UUT. Essentially, a device controller has to be synthesized in software. Such a test access routine is also a part of the system model and a subject of automatic synthesis based on the target UUT model and the instruction set of the CPU core.

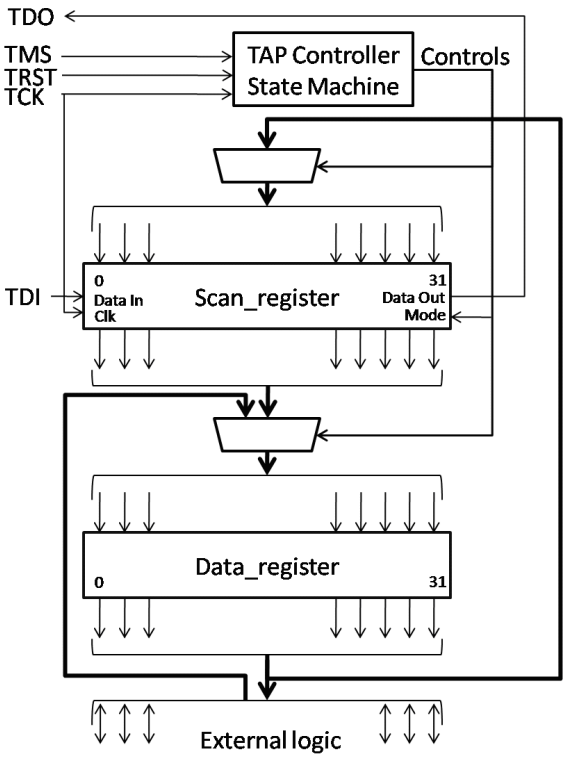

Figure 5: Data path for a part of IEEE 1149.1 TAP

Let us consider the structure depicted in Figure 5 as a simplified system to be modeled for test access purposes. Figure 5 presents a part of the standard *Test Access Port* TAP that consist of TAP controller state machine and scan register that is connected to respective data register. Data is shifted into the scan register through the serial *Test Data In* (TDI) bus when TAP controller state is "Shift-DR". The TAP controller "Controls" output is equal to 4 (Controls = 4) when the state is "Shift-DR". The load from the scan register into the data register is initiated when the TAP controller state is "Update-DR" (Controls = 8). Storing of data from the data register to the scan register is done when the TAP controller state is "Capture-DR" (Controls = 3). The TAP controller enters the reset state "Test-Logic-Reset" when the TRST signal is enabled. In the same state the data register obtains its reset value. The HLDD-model of the described structure (Figure 5) is shown in Figure 6.

Status of this activity: *DONE*.

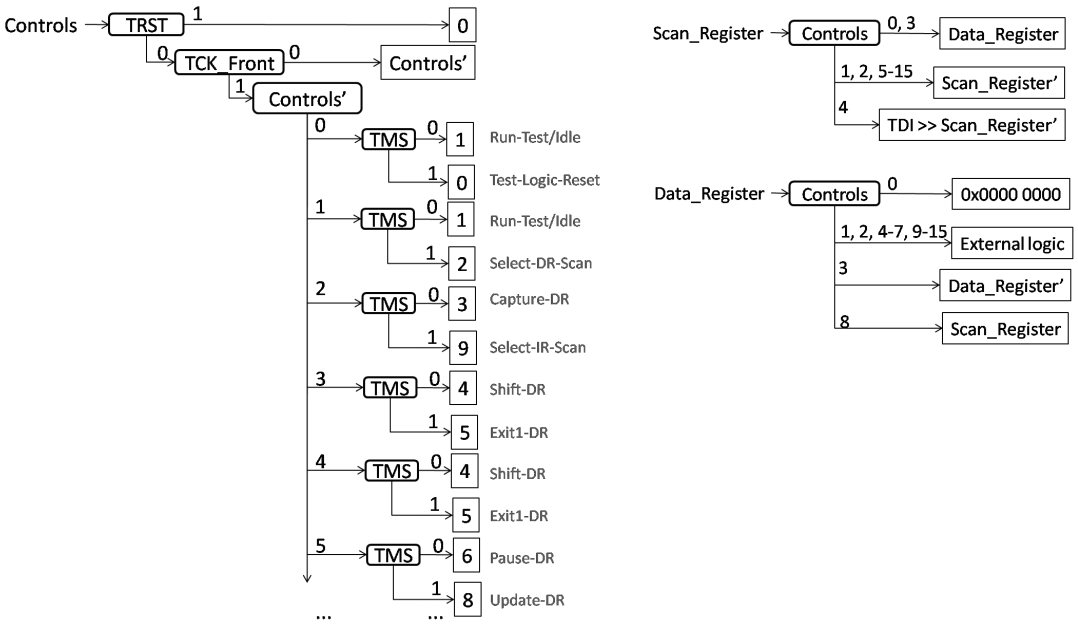

Figure 6: Model (HLDDs) for structure in Figure 5

5 Summary

A range of different types of reasoning engines is being developed within DIAMOND. While the general reasoning framework is independent of the application, demands on efficiency and quality of results require to adopt engines for the particular applications. The formal, semi-formal, and dynamic engines of DIAMOND cover all aspects from transaction level down to post-silicon and in-situ diagnosis. These engines address diagnosis, correction, and repair as planned in the *Description of Work* [9]. Applications for diagnosis and repair running on top of the reasoning engines are under development within WP2 and WP3, respectively.

6 References

[1] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *IEE/ACM International Conference on Automated Software Engineering*, pages 443–446. IEEE, 2008.

[3] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. LNCS 4963.

[4] DIAMOND Consortium. D1.3 - Status on the reasoning engines and dynamic techniques. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2010.

[5] DIAMOND Consortium. D2.1 - Transaction-level diagnosis. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2010.

[6] DIAMOND Consortium. D2.2a - Status on implementation-level diagnosis. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2010.

[7] DIAMOND Consortium. D2.3a - Status on post-silicon and in-situ diagnosis. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2010.

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

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

[10] DIAMOND Consortium. D1.2 - Definition of the diagnostic model. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2011.

[11] DIAMOND Consortium. D3.2a - Status on implementation-level correction. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2011.

[12] DIAMOND Consortium. D3.3a - Status on post-silicon and in-situ repair. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2011.

[13] B. Dutertre and L. De Moura. The Yices SMT solver, 2006. Available at http://yices.csl.sri.com/tool-paper.pdf.

[14] A. Finder and G. Fey. Evaluating debugging algorithms from a qualitative perspective. In *Proceedings of the Forum on Specification & Design Languages*, pages 37–42, 2010.

[15] A. Finder, A. Sülflow, and G. Fey. Latency analysis for sequential circuits. In *Proceedings of IEEE European Test Symposium*, pages 129–135, 2011.

[16] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli. Ground interpolation for the theory of equality. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 413–427. Springer Berlin / Heidelberg, 2009.

[17] G. Hofferek and R. Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In *International Conference on Formal Methods and Models for Codesign*, pages 31–42. IEEE, 2011.

[18] J.H.R. Jiang, H.P. Lin, and W.L. Hung. Interpolating functions from large boolean relations. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 779–784. ACM, 2009.

[19] S. Krishnaswamy, S. Plaza, I. L. Markov, and J. P. Hayes. Enhancing design robustness with reliability-aware resynthesis and logic simulation. In *International Conference on Computer-Aided Design*, pages 149–154, 2007.

[20] K.L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[22] A. Tsertov, A. Jutman, S. Devadze, and R. Ubar. Automatic soc level test path synthesis based on partial functional models. In *Proceedings of Asian Test Symposium*, 2011.

[23] A. Tsertov, A. Jutman, S. Devadze, and R. Ubar. Soc and board modeling for processor-centric board testing. In *Proceedings of Euromicro Conference on Digital System Design*, pages 575–582, 2011.

[24] T. Viilukas, J. Raik, M. Jenihhin, R. Ubar, and A. Krivenko. Constraint-based hierarchical untestability identification for synchronous sequential circuits. In *Proceedings of IEEE European Test Symposium*, pages 1–6, 2011.