FP7-ICT-2009-4-248613

DIAMOND

Diagnosis, Error Modelling and Correction for Reliable Systems Design

Instrument: Collaborative Project

Thematic Priority: Information and Communication Technologies

Definition of the Diagnostic Model (Deliverable D1.2)

Due date of deliverable: June 30, 2011
Actual submission date: June 30, 2011

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

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

Revision 1.7

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 D1.2 described in contract number 248613.

Table of Revisions

Version	Date	Description and reason	Author	Affected sections
1.0	February 24, 2011	Initial structure	G. Fey, R. Drechsler	All sections
1.1	April 7, 2011	Revised introduction and structure	A. Finder, A. Sülflow	All sections
1.2	April 19, 2011	TUG contribution	R. Könighofer, G. Hofferek	Sections 2.2, 3.1, and 4.1
1.3	April 28, 2011	LiU contribution	U. Ingelsson	Sections 2.3, 3.3, 4.2, 4.3, 5.2 and 5.3
1.4	May 8, 2011	IBM contribution	E. Arbel	Sections 2.4, and 4.3
1.5	May 11, 2011	TEDA contribution	S. Scholefield	Section 5.4
1.6	May 23, 2011	Preparation for internal review	G. Fey, A. Finder, A. Sülflow	All sections
1.7	May 31, 2011	Internal review changes	E. Arbel	All sections

Authors, Beneficiary

Rolf Drechsler, University of Bremen
Görschwin Fey, University of Bremen
Alexander Finder, University of Bremen
André Sülflow, University of Bremen
Robert Könighofer, Graz University of Technology
Georg Hofferek, Graz University of Technology
Urban Ingelsson, Linköping University
Eli Arbel, IBM
Jaan Raik, Tallinn University of Technology
Stephen Scholefield, TransEDA Systems Ltd
Artur Jutman, Testonica Lab

Executive Summary

This document presents an overview of the holistic diagnostic model applied 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, in order to capture the individual requirements by the different applications. The present deliverable describes the properties of the diagnostic model.

List of Abbreviations

ATPG - Automated Test Pattern Generation
CPU - Central Processing Unit
CDFG - Control and Data Flow Graph
DoW - Description of Work
DUV - Design under Verification
ESL - Electronic System Level
FP7 - European Union's 7^{th} Framework Programme
HDL - Hardware Description Language
HLDD - High-Level Decision Diagram
IC - Integrated Circuit
IP-Core - Intellectual Property Core
LTL - Linear Temporal Logic
PSL - Property Specification Language
RTL - Register Transfer Level
SMT - Satisfiability Modulo Theories
TLM - Transaction Level Modeling
WP - Work package

Table of Contents

1 Introduction and Overview

DIAMOND targets different application domains at different levels in the design flow. The holistic diagnostic model can be considered the "backbone" of the DIAMOND infrastructure. The model connects the application level to the reasoning engines as described in Deliverable D1.1 [4]. Figure 1 gives a general overview of DIAMOND. The holistic diagnostic model is developed within Task T1.1 of WP1 and depends partially on the requirements and end-user needs described in Deliverable D4.1 [8]. Figure 2, taken from the *Description of Work* [9], shows in more detail how the diagnostic model couples the application domains – diagnosis and correction – with the underlying reasoning engines. The status of the reasoning engines has been described in Deliverable D1.3 [5]. The status of the applications is considered by Deliverable D2.2a [6] and Deliverable D2.3a [7] that describe the status on implementation-level diagnosis and the status on post-silicon and in-situ diagnosis, respectively.

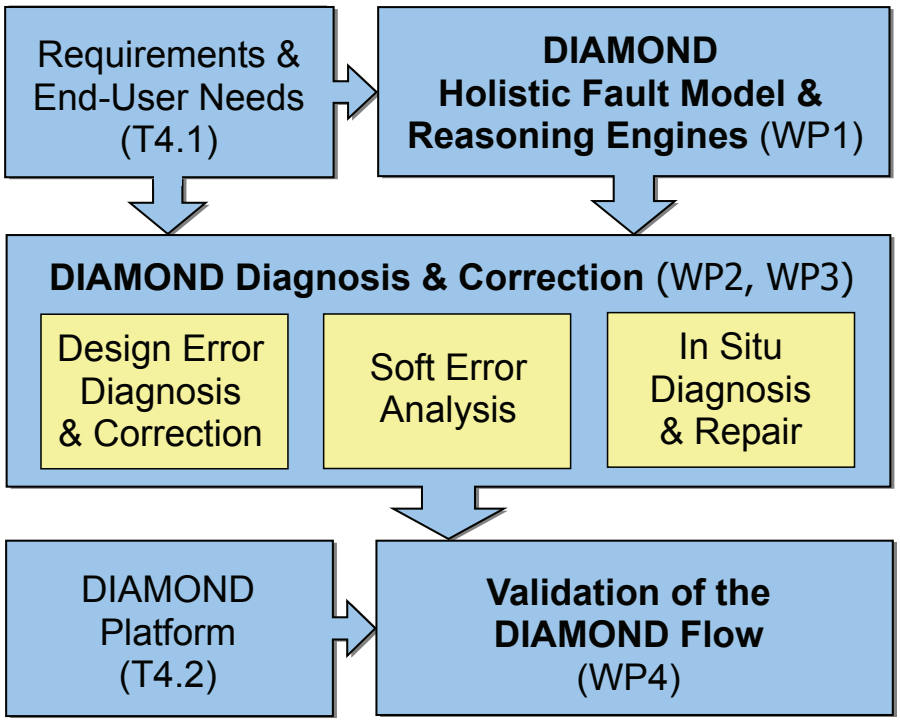

Figure 1: Overview of DIAMOND

The present deliverable describes the diagnostic model itself as a result of reaching Milestone 1.1 *Common Data Structure for the Diagnostic Model*. The common data structure for the diagnostic model has been developed by all partners and is an abstraction of the diverse implementations, while individual implementations of this abstraction are distributed between partners. This is justified by several administrative and practical reasons: (a) some partners have private code, e.g., com-

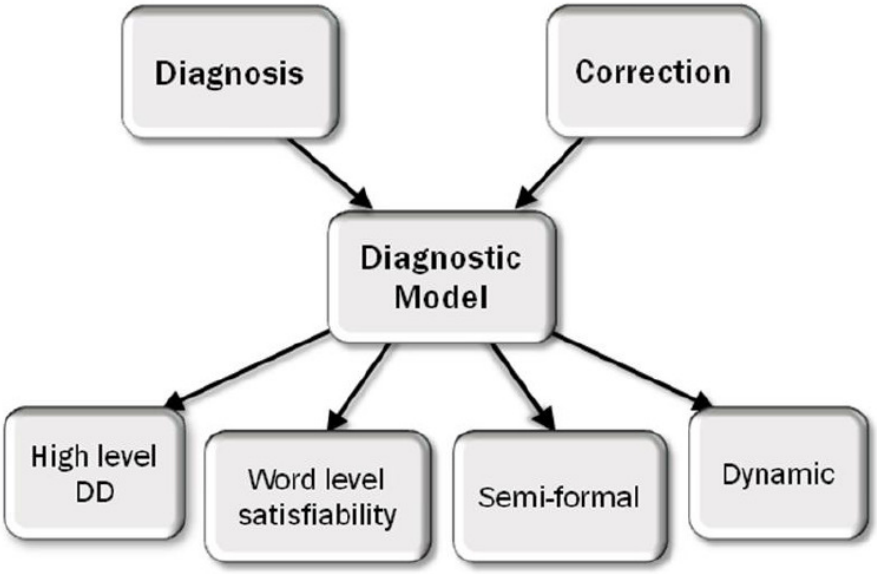

Figure 2: Diagnostic model within DIAMOND

panies, (b) some partners use an integration to legacy tools that where available before DIAMOND started, and (c) the overhead for integrating the different implementations that typically focus on certain aspects of the diagnostic model would not yield any advance in theory or practice. Instead unifying the diagnostic model into a holistic abstraction shows how the various analysis tasks are related and how multiple abstraction levels as well as applications can be treated by common algorithms. By this, potential for advancing theory or practice becomes visible and will be exploited while DIAMOND progresses.

There exist several successful approaches that show how various analysis tasks are related and how different abstraction levels can be treated by common algorithms:

- Latency analysis investigates both the effect of design errors and the effect of soft errors in the corresponding abstraction level of designs.
- A fault management approach developed jointly by multiple partners handles diagnosis and repair in the same way, regardless of whether the error is on system-level, component-level, or gate-level, with varying level of efficiency depending on the abstraction level.
- The High-Level Decision Diagram (HLDD) data structure is used both at implementation level and at in-situ diagnosis. The HLDD-based critical path tracing engine has been applied to design error localization and to calculation of critical soft-error lists.
- At the system-level, the tool FoREnSiC is under development which includes a common flow chart data structure for various diagnosis and correction tasks.

Figure 3 conceptually shows the diagnostic model. The coarse structure is defined by the overall application as shown on the left: based on some input data, the algorithm is executed and returns some output to be used and interpreted by a designer.

The data structures within the diagnostic model corresponding to these steps are shown in the box. The input is given by information specifying the problem, e.g., information on the implementation of the circuit and the specification of expected behavior. The algorithms within DIAMOND are based on reasoning engines that need certain input parameters like the type of the application, i.e., localization or correction, and on the types of faults to be addressed. Finally, the output may be represented in various ways, depending on the application. For instance, fault candidates may be returned from localization, while suggestions for repairs may be returned by a correction algorithm. All of this information is stored within the diagnostic model.

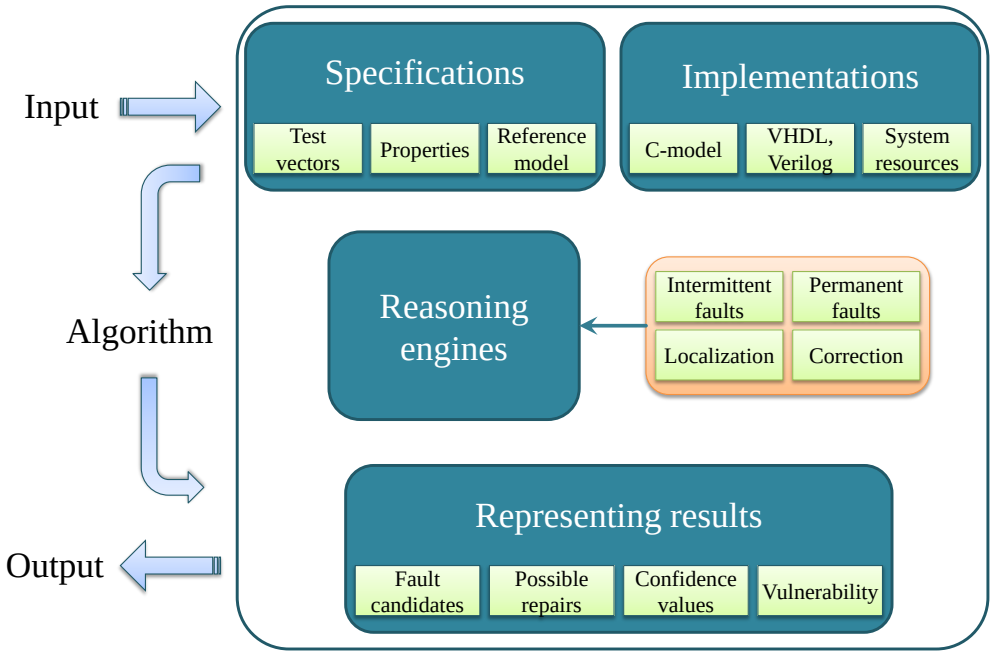

Figure 3: Insight into the diagnostic model

The rest of this document is also organized along the structure of the diagnostic model: Sections 2 and 3 describe the different types of specifications and implementations, respectively, required at different abstraction levels. Section 4 explains the holistic fault model. The infrastructure for representing results from diagnosis and correction is described in Section 5. A summary is given in Section 6.

2 Describing Specifications

For diagnosis or correction a specification is required in order to differentiate desired behavior from incorrect behavior. The specification of intended behavior of a design is defined by a formal description. These specifications are not required to fully describe all desired behavior. For example, assertions or test vectors partially describe the desired behavior, whereas a reference implementation gives a complete description of the expected behavior. The requirements on the completeness and the level of abstraction depend on the application. In the following, the different types of specifications and their application domains as considered within DIAMOND are introduced.

Section 2.1 describes CPU models used as an abstract way to specify in-situ tests. Section 2.2 introduces formal specification as a general way to formalize desired behavior as a starting point for debugging. Trace-based specifications are considered in Section 2.3 as an incomplete but easily accessible specification. Finally, resilience against faults occurring in field may be required and therefore be specified as described by Section 2.4.

2.1 Reference Models

The goal of CPU modeling methods that are developed within DIAMOND is to provide automated test and diagnostic access to the particular components of a target system. In the proposed scheme, a CPU core is used for running test and diagnosis routines (apply diagnostic stimuli and capture responses) downloaded from an external tester, thus playing the role of an internal (in-system) tester.

The reference model is not intended to fully describe the behavior of microprocessor and peripheral modules, but rather to provide a semi-formal specification of a small subset of CPU/peripheral functional blocks. However this specification should contain enough information in order to automatically establish diagnostic access to desired components. The mathematical basis for the model is the theory of High Level Decision Diagrams (HLDDs) [15].

The diagnostic routines are typically fed trough the CPUs/SoCs debug port either to the embedded memory or to the external memory of the system. In both cases the access to the memories is provided by the means of the CPU/SoCs infrastructure including memory controllers and bus matrices. The full data path between the external test controller and the component under diagnosis includes debug port, CPU core, firmware running on the CPU, peripheral controllers, interconnection structure between CPU, and target component (see Figure 4).

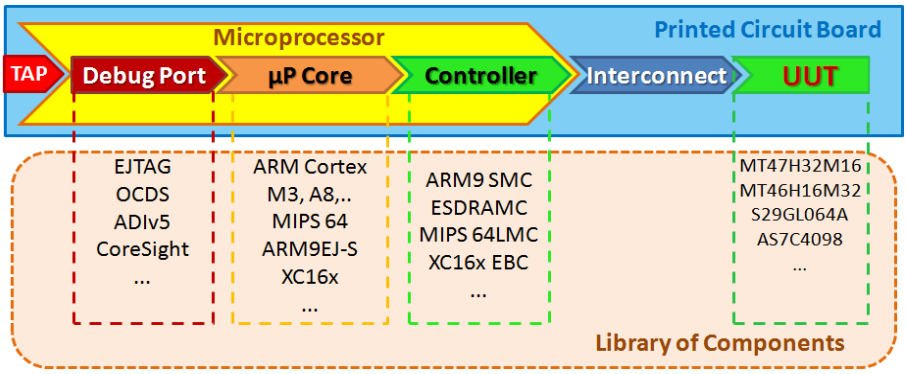

Figure 4: Full data path between external test controller and component under diagnosis

Along the whole diagnostic path the same modeling concept is used. This approach allows building a unified model with the ultimate goal of automatic generation and feeding a sequence of test data trough the propagation chain mentioned above. The key idea behind the proposed concept is to represent the system as a set of tightly interrelated models. All such models can be combined together block by block to represent the complete structure of a target system. Some model components (e.g. debug port functionality) can be described at the behavioral (functional) level, while others may also require structural (e.g. interconnect) or timing information (e.g. memory devices, controllers).

A typical model of the system would contain only those components, functional blocks, busses, ports etc., that are needed to be activated during propagation of test and diagnostic data. We distinguish three groups of sub-models that form the final specification of system:

1. The first group contains blocks that describe the functionality of the internal modules of a CPU (such as CPU core, JTAG, debug module, bus matrix, external bus interface, set of memory controllers and peripheral device controllers).
2. The second group describes the functionality of a component which is diagnosed and encloses data for automatic initialization or configuration of this component. For instance, this group can contain memory initialization procedure, write access and read access protocols, configuration and timing parameters (refresh rates, operating voltages, etc.).
3. The third group represents the information about the interconnection of different components in the system. This information could be automatically extracted from the netlist files.

The sub-models of the first two groups are built using a semi-automated approach, where the information from the documentation is converted by an engineer into a special readable description language and then automatically transformed to a system of HLDD graphs. The information of the third group could be automatically extracted from the netlist files.

As it was mentioned above the goal of building a CPU model is to provide automated synthesis of the routines for test and diagnostic access. The input to the automation procedure is a unified model that contains descriptions of CPU/peripheral

modules, component under diagnosis, and interconnection information. The output is the synthesized test program. The central element of the test automation procedure is a constraint solver. The constraint solver tries to find the path through the system of HLDD graphs that starts from the target component, goes to the corresponding peripheral modules of CPU, follows through processor core to its debug module, and finishes on the external debug port (JTAG). In the end, the synthesized test program will contain a sequence of stimuli that should be applied to the external debug port for providing diagnostic access to desired components.

These reference models are a specific structure to specify the behavior of test infrastructure in a condensed way. The following formal specifications are a more general way of specification not tied to a certain domain.

2.2 Formal Specifications

Another way to describe the desired behavior of an implementation is to use a formal specification. The word *formal* means that the specification language has a strictly defined semantics. Formal specifications provide a high flexibility regarding their properties and applications. In contrast to reference models, they may be (but do not need to be) incomplete. In contrast to trace-based specifications, they do not enumerate desired behavior for different executions. Instead they define properties that have to hold for all executions and inputs, thus providing the potential for higher coverage in verification. Formal specifications can be checked both dynamically (for a concrete execution) and statically (for all executions simultaneously using tools like model checkers). Detected misbehavior is typically returned as a sequence of input values for which the implementation violates the specification. Besides verification, formal specifications are also used to unambiguously communicate design intents. This avoids misunderstandings, e.g., between collaborating designers. Furthermore, they are used in property synthesis, where a provably correct implementation is created automatically from its specification.

A simple and widely used way to specify behavior is to include assertions in the implementation. An assertion is a predicate that is supposed to hold in every execution at the location where it is placed. In their simplest form, assertions are formulated in terms of features provided by the implementation language itself. Many modern programming languages also provide mechanisms to check assertions at runtime. An example is the `assert()`-macro of the programming language C. The macro takes an arbitrary condition formulated in C-syntax as argument. If the program is compiled with assertions enabled, the condition is evaluated whenever the execution reaches the assertion. If it evaluates to false, the program is aborted with an error message pinpointing the location of the assertion violation.

Another widely used formal specification language is the Property Specification Language (PSL) [1, 10]. PSL can be seen as an extension of Linear Temporal Logic (LTL) [13]. Like LTL, PSL also provides temporal operators that allow

specifying the timing of events. This does not only include cycle accurate descriptions but also more general statements like, for instance, that some condition has to be fulfilled eventually in the future (without saying exactly when), that some predicate has to hold infinitely often in an infinite execution, or simply that it has to hold always or never. This makes PSL perfectly suitable for specifying reactive hardware designs. PSL became an IEEE standard (IEEE 1850 Standard for Property Specification Language) in 2005.

In DIAMOND we are working with different kinds of formal specifications. For instance, C-assertions are used to specify transaction-level models. They are also supported as specification by the tool FoREnSiC, developed within Task 3.1. In Task 2.1 we address debugging of formal specifications [12]. A debugging method has been implemented for (a subset of) PSL in the tool RATSY [2].

2.3 Trace-based Specifications

In some cases a formal specification defined on all input sequences may not be available. For instance, this is the case in a simulation based validation flow running the implementation against a test-bench. In such a case, a set of test vectors may be used to partially describe the expected behavior. Due to the large size of the search space, trace-based specifications typically suffer from low coverage and correctness cannot be decided for the remaining input sequences. On the other hand, test traces can be captured from post-silicon tests as well as from simulation-based verification approaches. Thus, such a specification may be accessible more easily than a formal specification.

A test trace is defined by a sequence of input assignments (i.e. a trace) together with expected output responses of the *Design Under Verification* (DUV). Using test traces, the DUV is verified by simulating all test traces and checking the equivalence to the expected responses. Detected misbehavior is returned as a set of failing test traces that are considered as input for fault localization and repair algorithms.

Using such a test trace as discussed above, this test trace could also potentially be used for post-silicon test and diagnosis, reusing the test-bench logic for applying and evaluating the test to narrow down fault candidates. This would serve as a complement to deterministic test data, generated by a commercial ATPG tool for the design once it has been verified. Conceptually, the test trace from verification and the test bench logic could also be re-used for in-situ test and diagnosis in the context of embedding deterministic test data on the system containing the design, such as described in Section 4.2 of Deliverable D2.3a [7]. Currently, only test data determined by commercial ATPG tools is used for trace-based specifications applied to in-situ test and diagnosis within DIAMOND.

The diagnostic model explicitly holds the test traces together with the correct expected output responses.

2.4 Expected Error Detection and Correction

In addition to describing how the design should behave under normal operating conditions, i.e. where the hardware works without any faults, specifications should also describe design behavior in the presence of errors, such as those originated by soft error hits. While design behavior can be described either by some formal temporal language or by trace-based specifications, as described above, this is not usually the case when describing error detection and correction behavior.

In order to specify built-in error detection and correction in the design, designers have to first identify the latches which are important for correct design functionality, that is those latches which will probably cause silent data corruption or a machine hang if hit by soft errors. For example, a program counter register may be defined as such. Next, a decision has to be made regarding which action to take in the case of a soft error hit on any given latch or register. Such decision may involve choosing to protect latches with self correcting mechanisms, such as ECC in register files, initiate recovery actions by restarting the machine to some previously stored checkpoint state, or checkstopping the design, i.e., moving the design into a safe state.

The process described above may result in a list of latches, each of which is assigned with the required error detection and/or correction mechanism. Alternatively, a list of error checkers which will be used in the implementation may also be specified, in the form of exact names or by defining naming conventions for describing different types of detection and correction mechanisms. This specification can be used later on to verify whether different error protection metrics are met, such as percentage of checked latches and type of checking for each. In addition, since recovery actions usually involve reseting the design to some known state by initializing a predefined set of latches, any formal language which supports propositional logic can be used in order to specify latch behavior during recovery actions.

Once a specification for the expected error detection and correction behavior is created, automated techniques can be used to verify whether the implementation meets the error detection and correction expectations. For example, dynamic verification techniques, namely using error injection in simulation, can verify that soft errors on protected latches are caught and, if appropriate, corrected automatically by the implementation. Another use for error detection and correction specification is to verify that there are no latches which are highly vulnerable to soft error hits and are not protected in the implementation.

3 Describing Implementations

During the design process of ICs the implementation progresses through several levels of abstraction in a refinement flow. In this section, different abstraction levels that are considered within DIAMOND and their storage within the diagnostic model are described.

In more detail these are the transaction level in Section 3.1, the *Register Transfer Level* (RTL) in Section 3.2, and post silicon descriptions in Section 3.3.

3.1 Transaction Level

The first models of a hardware design are typically not written in hardware description languages but rather defined on a higher abstraction level, namely the transaction level. In *Transaction Level Modeling* (TLM) [3], details about the components are separated from the details about their communication. Communication is modeled via channels. Channels abstract away low-level details of the communication (e.g., details about buffers, busses, etc.) and focus on modeling functionality (what is transfered between which components). Similar terms, which are often used synonymously with TLM, are System Level Modeling or Electronic System Level (ESL) Modeling. The core idea is the same: the focus is on modeling functionality and algorithmic structure while omitting low-level details like exact timing, bitwidths, communication details and so on. In this document, these terms are used synonymously as well.

On the transaction level, designs are typically modeled as software programs. A program consists of functions, which in turn consist of statements. Functions model different modules of the design or parts of the algorithm. Function calls can be used to model communication between modules. Information can be passed via parameters and return values or global data structures. Typical languages for transaction level modeling are C, C++, or SystemC, which is an extension of C++ with a simulation kernel. These languages provide rich features like, for instance, dynamic memory allocation, recursion, and a large set of available libraries, which allow a design to quickly explore different design options on a high abstraction level early in the development process. Since transaction level models are software programs and thus executable, they can also be analyzed and dynamically verified. This reduces the chances that suboptimal design decisions or conceptional errors are uncovered only after the implementation work has been done. Another advantage of higher level models is that execution is typically much faster than simulation on lower abstraction levels.

The diagnostic model stores transaction level implementations in terms of *Control and Data Flow Graphs* (CDFG). For instance, the CDFG can then be refined to a word-level netlist as used for implementations on the RTL.

3.2 Register Transfer Level

An implementation in RTL is a refinement of an implementation on transaction level. Abstractions in RTL define the behavior of the circuit by specifying the signal flow between registers and logical operations.

Typically, a description in RTL is given as source code in a formal HDL. This is the basis for synthesizing logic level netlists in the subsequent design steps. HDLs enable implementing a cycle accurate model of a circuit that can be used for simulation to explore the behavior of the design. Alternatively, formal verification is frequently applied on RTL. Two well-known representatives of HDLs are *Verilog* and *VHDL*.

Most HDLs support Boolean and arithmetic operations to model the signal flow, whereas registers are used to model the temporal behavior of the circuit. The operations and registers are defined in modules which are ordered hierarchically. Therefore, a single module may be instantiated multiple times in the same design. Moreover, IP-Cores are often used to reduce time-to-market.

A simulator gives insight into the behavior of the design. While simulating a trace, values of signals at particular time steps become observable and enable analyzing the behavior of the implementation in detail. Due to the good observability of the internal behavior of the implementation given on RTL, specific analysis tasks like fault localization and in-situ fault analysis are supported.

The diagnostic model stores HDL implementations in terms of word-level netlists. A simple synthesis is performed to derive the netlist from the HDL description. All functional operations are synthesized into combinational circuitry and into storage elements as inferred from the HDL description. No optimizations are performed in this synthesis step to keep a one-to-one correspondence of netlist elements to parts of the HDL description. The mapping between netlist and the original HDL source is also stored to allow back-annotation of results to the source level.

3.3 Post-Silicon

A circuit layout would accurately describe hardware at the post-silicon stage. However, the localization and correction algorithms should not depend on such detailed descriptions. Instead a certain abstraction is required to be comprehensible for a designer or to be suitable for taking counter-measures after detecting faults.

Therefore, a system in post-silicon is described as the set of resources it contains (processing elements, memories, etc.) and structural information about how each resource can be accessed from the outside of the system. In the diagnostic model, a resource manager is associated to each system. Such a resource manager maintains a record of the status for each resource as a resource map and a system health map (see Figure 5). The resources may be idle, busy or fault marked. A fault marked resource is isolated from the functional operation of the system and no tasks are scheduled on a fault marked resource. This information is maintained in the resource map. The system health map keeps error statistics for each resource. Further details on diagnosis data and repair data are given in Section 5.2 and Section 5.3.

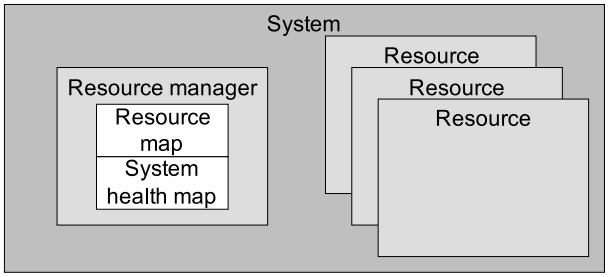

Figure 5: System overview

Most in-situ diagnosis activities depend on a test set, consisting of test stimuli inputs and expected test responses for identifying the faulty component. This test set is pre-generated, based on gate-level descriptions of the system components. Gate-level descriptions are refinements of RTL descriptions that are discussed in Section 3.2. The test sets are generated from gate-level descriptions by state-of-the-art ATPG tools. For each resource, the diagnostic model contains such a test set.

4 Modeling Faults

Section 3 defines different levels of abstraction of an implementation. For different purposes different types of faults have to be modeled. For example, design bugs on the transaction level are different from transient faults in the actual chip. All these different types of faults are stored in the diagnostic model allowing to handle them by similar algorithms.

This section provides a holistic fault model to embed different types of faults independent of the abstraction level and for all the different applications considered within DIAMOND. *Components* are used to describe faulty parts of an implementation and may consist of, e.g., a single gate, a set of gates, or a statement in source code. Thereby, a fault may be permanent (i.e. a component behaves faulty at *all* time frames) or a fault may be intermittent (i.e. a component behaves faulty at *some but not all* time frames). In this context, transient faults resulting from radiation effects are intermittent faults restricted to a very short time frame. Moreover, multiple components may be faulty simultaneously. Each of the following sections defines its own component model to describe the specific faults in more detail.

The following description differentiates between design bugs in Section 4.1, permanent physical faults in Section 4.2, and intermittent physical faults in Section 4.3.

4.1 Design Bugs

Design bugs are permanent. Therefore, an activation of the bug by a certain input condition always results in the same faulty values that may propagate to an observable point.

On the transaction level, designs are modeled as software programs. A program consists of functions, which in turn consist of statements, and statements consist of expressions. Hence, on the transaction level, a component may be

- a function,
- a statement, or
- an expression.

Typically, an RTL description consists of a set of modules that are ordered hierarchically. The behavior of a single module is described by a set of statements defining control flow and data flow.

According to this, the portion of the RTL design affected by a bug is a set of statements or modules. Typical components considered for diagnosis or correction at this level are:

- modules,
- statements, or
- gates.

Typically, a more fine-grained component definition allows for more detailed error localization and more focused error correction. On the other hand, a finer granularity also increases the search space of possibly erroneous components. Hence, there is no optimal component definition for design bugs. But there is a trade-off between accuracy and efficiency.

4.2 Permanent Physical Faults

Permanent physical faults are different from bugs in the sense that permanent physical faults will not affect all instances of the design when manufactured in silicon as bugs do. Furthermore, permanent physical faults may be introduced in manufacturing or in-field on silicon that has passed manufacturing test.

Permanent physical faults can affect the logic, memory elements and interfaces. In Deliverable D2.3a [7], the following categorization of permanent faults was presented.

A permanent physical fault in logic causes a faulty timing behavior or a faulty logic behavior that may propagate to an observable point for a certain input sequence. During in-situ monitoring, permanent physical faults may manifest as intermittent, however at off-line test they should be detected as permanent.

A permanent physical fault in memories may be detected through parity errors, or infinite ECC repair rate. In both cases a fault counter is required, e.g. leaky bucket, which should be considered as an instrument.

A permanent physical fault in interfaces should manifest itself as permanent during in-situ monitoring. Off-line tests of the interface should confirm this. There are several ways to analyze the potential cause, e.g. loss of signal, loss of frame synchronization, check-sum error, etc.

The resulting component model is composed of:

- memories
- interfaces
- logic structures

4.3 Intermittent Physical Faults

Transient faults lead to soft errors which represent a common cause for intermittent physical faults. Traditionally, soft errors were modeled only for latches or registers. However, in sub-micron technologies logic gates also become dominant from the soft error vulnerability point of view. Hence, typical components considered for diagnosis and correction in this level are:

- a register
- a signal

The more general intermittent faults on the physical level can be modeled by adding a special logic structure for these types of components and other components affected by them. In the RTL level, intermittent faults caused by soft errors can be modeled by adding multiplexing logic which is used to invert a signal or a register value when a soft error hit is simulated for that component. Furthermore, single event upsets, a common model for soft error hits, can also be modeled using this approach by restricting the number of signal or register bit flips to only one at each time frame. This is achieved by using a one-hot scheme on the multiplexer selector bits used to model the soft errors.

In Deliverable D2.3a [7], the following categorization of intermittent faults was presented. Similar to permanent faults, intermittent physical faults can affect logic, memories, and interfaces. Intermittent physical faults in logic, memory, or interfaces may not be detected in off-line tests. A fault counter, e.g. leaky bucket, is required to identify an intermittent physical fault in logic, memory, or interfaces. Exceeding a threshold for the fault counter should be regarded as a permanent fault.

5 Representing Results

The results of localization and correction algorithms are returned on the component model. For this purpose back-annotation from the algorithmic level to the problem description is required as described in Section 5.1. Results as returned for diagnosis and correction are described in Section 5.2 and Section 5.3, respectively. Finally, an integration with a work-flow engine allows analyzing the different options returned by the algorithms as detailed in Section 5.4.

5.1 Back-annotation

The algorithms running on top of the reasoning engines do not directly operate on the initial representation of the problem. Instead multiple transformations or synthesis steps typically precede the application of the reasoning engine. Nonetheless, the results of diagnosis and correction algorithms have to be represented on the same level as the initial problem description. That is, the results on the algorithmic level have to be back-annotated to the initial description. For this purpose a mapping structure is kept for relating data structures representing the problem in the algorithm to the original description. For instance, in case of using solvers for *Satisfiability Modulo Theories* (SMT) the low level constraints can be related to HDL source code when debugging RTL implementations.

5.2 Diagnosis Data

Diagnosis and localization are applied on various abstraction levels and on different types of faults. The diagnosis results depends on these parameters.

- For pre-silicon diagnosis – transaction level or implementation level debugging – a fault candidate is a pair of a fault type and a component as described in Section 4. To rank the set of fault candidates confidence values may be computed which represent probabilities that a component is faulty, i.e. a fault candidate.
- To assess whether a design handles transient faults as specified (see Section 2.4). For this purpose vulnerability and latency values for a DUV may be computed and related to components. Vulnerability analysis defines and

classifies critical points in a system and thus shows ways which part of a system should be made more robust. Latency analysis computes for how many cycles a sequential circuit has to be analyzed at least or at the maximum.

- For post-silicon diagnosis, a fault candidate is a pair of a fault type and a logic gate or an interconnect which may belong to embedded logic in an IC. The confidence values for fault candidates will depend on the approach to diagnosis, of which several exist in literature [11, 14, 16].
- For in-field diagnosis, a fault candidate is the identifier of a replaceable or fault-markable component, and this will depend from design to design. A resource map detailing such components in the system will be used for fault-marking and book-keeping of fault candidates. The fault type is permanent physical fault.

  A system health map, consisting of an indicator representing fault marking and a counter for transient physical faults for each component in the resource map, will be maintained.

  Examples illustrating a resource map and a system health map are given in Table 1 and Table 2 respectively. A resource has one out of three states, namely Idle, Active and Faulty. The status of a resource is given in the resource map. The system health map details the number of soft errors that have been detected in a resource since the system was first brought into operation.

Table 1: Example resource map

Resource ID	**Status**
Resource 1	Faulty
Resource 2	Idle
Resource 3	Active
Resource 4	Active
Resource 5	Faulty
...	...
Resource n	Idle

Table 2: Example system health map

Resource ID	**Number of errors**
Resource 1	0
Resource 2	5
Resource 3	0
Resource 4	15
Resource 5	17
...	...
Resource n	0

5.3 Correction Data and Repair Data

Design bugs are corrected while physical faults are repaired.

A repair of a design bug in pre-silicon is the replacement of a faulty component by another component fixing the design bug. For instance, a faulty statement in a C-model is replaced by a correct one. For this purpose functional replacements are annotated to components as a correction of a design bug.

For in-field repair, a repair to recover from a transient physical fault will be represented by the command to re-execute the affected task. A repair to recover from a permanent physical fault will be represented by the command to re-execute the

affected task on another processing element, if redundant processing elements are available in the system.

5.4 Workflow

The workflow described in the following is not an integral part of the diagnostic model, but shows one way of integrating with the standard design flow. This workflow enables a designer to utilize the results stored in the diagnostic model. The workflow itself is defined and implemented within WP 2 and WP 3 – only the overall idea is described in this section.

The approach is to store the results in a relational database management system (RDBMS) to scale up the amount of tests which can be automatically generated and re-run. Each code change is recorded so that automatic error correction techniques can be applied.

Based on this engine a designer can be enabled to analyze the suggested diagnostic results and potential corrections one after another by applying the related code change and undoing the change if the result is not satisfactory.

6 Summary

In this deliverable the diagnostic model of the DIAMOND project is described as a whole. Different views are integrated with respect to different applications. This includes different descriptions for specifications and implementations as well as different abstraction levels and applications. Furthermore, various types of faults are addressed and results are stored for further use by a designer. The diagnostic model unifies all of this information conceptually for storage within a single data structure to be used by different reasoning engines.

7 References

[1] Accellera. *Accelera Property Specification Language: Reference Manual — Version 1.0*.

[2] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. Ratsy - a new requirements analysis tool with synthesis. In *Computer Aided Verification*, volume 6174 of *LNCS*. Springer-Verlag, 2010.

[3] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24. ACM, 2003.

[4] DIAMOND Consortium. D1.1 - Requirements and concept of the diagnostic model. DIAMOND - Diagnosis, Error Modeling and Correction for Reliable System Design, ICT 2009.3.2, 2010.

[5] 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.

[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 end-user requirements. 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] C. Eisner and D. Fisman. *A practical introduction to PSL*. Springer-Verlag New York Inc, 2006.

[11] S. Holst and H.-J. Wunderlich. Adaptive Debug and Diagnosis Without Fault Dictionaries. In *Proceedings of the IEEE European Test Symposium (ETS)*, 2007.

[12] R. Könighofer, G. Hofferek, and R. Bloem. Debugging unrealizable specifications with model-based diagnosis. In *Proceedings of the Haifa Verification Conference*, LNCS, pages 29–45. Springer-Verlag, 2010.

[13] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[14] M. A. Shukoor and V. D. Agrawal. Compation of Diagnostic Test Set for a Full-Response Dictionary. In *Proceedings of the North Atlantic Test Workshop*, 2009.

[15] R. Ubar, J. Raik, A. Karputkin, and M. Tombak. Synthesis of high-level decision diagrams for functional test pattern generation. In *Proceedings of the 16th International Conference on Mixed Design of Integrated Circuits Systems (MIXDES)*, pages 519 –524, 2009.

[16] W. Zou, W.-T. Cheng, and S. M. Reddy. Bridge Defect Diagnosis with Physical Information. In *Proceedings of the IEEE Asian Test Symposium*, pages 248–253, 2005.