FP7-ICT-2009-4-248613

DIAMOND

Diagnosis, Error Modelling and Correction for Reliable Systems Design

Instrument: Collaborative Project

Thematic Priority: Information and Communication Technologies

Transaction-Level Diagnosis (Deliverable D2.1)

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: Graz University of Technology

Revision 1.6

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 D2.1 described in contract number 248613.

© Copyright DIAMOND 2011. All rights reserved.

Table of Revisions

Version	Date	Description and reason	Author	Affected sections
1.0	Sept. 14, 2011	Initial structure	R. Könighofer, G. Hofferek	All sections
1.1	Oct. 13, 2011	TUG contribution	R. Könighofer	Section 2.1 and 3.1
1.2	Oct. 17, 2011	Initial LiU contribution	E. Larsson	Introduction, Section 5.1
1.3	Oct. 19, 2011	UNIB contribution	A. Finder, G. Fey	Section 2.2 and 4.2
1.4	Oct. 29, 2011	Updated LiU contribution	E. Larsson	Section 5.1
1.5	Nov. 16, 2011	TUT contribution	J. Raik	Section 4.1
1.6	Nov. 23, 2011	Minor improvements to all sections.	R. Könighofer, G. Hofferek	All sections

Authors, Beneficiary

Georg Hofferek, Graz University of Technology
Robert Könighofer, Graz University of Technology
Görschwin Fey, University of Bremen
Alexander Finder, University of Bremen
Erik Larsson, Linköping Universitet
Urmas Repinski, Tallinn University of Technology
Jaan Raik, Tallinn University of Technology

Executive Summary

This document summarizes the main achievements in Task T2.1 *Transaction-Level Diagnosis*, which is part of WP2 in the DIAMOND project. It is the final deliverable for this task. Together with the deliverables D2.2b and D2.3b, it contributes to milestone M2.2 *Prototypes of diagnosis tools implemented*.

Various formal, semi-formal and dynamic techniques for error localization on the transaction-level are presented. This includes techniques to diagnose over-constraint formal specifications, to diagnose inconsistencies between hardware descriptions and their transaction-level specification, techniques based on symbolic execution, and techniques using program slicing. Moreover, a fault-management infrastructure for multi-processor system-on-chip architectures is described. Finally, the tool FoREnSiC implementing several of the presented debugging techniques is briefly introduced.

List of Abbreviations

ALU - Arithmetic Logic Unit
API - Application Programming Interface
CPU - Central Processing Unit
CTRL - Control
DSP - Digital Signal Processor
EIF - Error Indication Flag
FIM - Fault Injection Manager
FIPI - Failure Indication and Propagation Infrastructure
FSM - Finite State Machine
GCC - GNU Compiler Collection
IAI - Instrument Access Infrastructure
IM - Instrument Manager
LHS - Left-Hand Side
MPSOC - Multi-Processor System-on-Chip
PC - Program Counter
RHS - Right-Hand Side
RM - Resource Manager
RTL - Register Transfer Level
SIB - Segment Insertion Bit
SMT - Satisfiability Modulo Theories
SSA - Static Single Assignment
WP - Work package

Table of Contents

Table of Revisions iii
Authors, Beneficiary iii
Executive Summary iii
List of Abbreviations iv
Table of Contents v
1 Introduction and Overview 1
1.1 Overview of the Methods and the Document 2
1.2 Link to the DoW 3
2 Formal Diagnosis Methods 5
2.1 Activity: Diagnosis of Formal Specifications 5
2.2 Activity: Diagnosis via RT-Level techniques 7
3 Semi-Formal Methods 9
3.1 Activity: Diagnosis using Symbolic Execution 9
4 Dynamic Methods 13
4.1 Activity: Diagnosis via Dynamic Slicing 13
4.2 Activity: Simulation-Based Debugging of SystemC 14
5 Integration and Implementation 17
5.1 Activity: Fault Management Infrastructure 17
5.2 Activity: FoREnSiC 18
6 Summary 23
7 References 25

1 Introduction and Overview

DIAMOND addresses automated error localization and correction on different levels in the hardware design flow. Fig. 1 illustrates how this is achieved. Based on end-user requirements, identified in Task T4.1 and summarized in Deliverable D4.1 [5], a holistic diagnostic model is developed in WP1. This diagnostic model, presented in Deliverable D1.2 [7], is the backbone of the DIAMOND infrastructure. Together with the reasoning engines developed in WP1, it forms the basis for error localization and correction, addressed in WP2 and WP3, respectively.

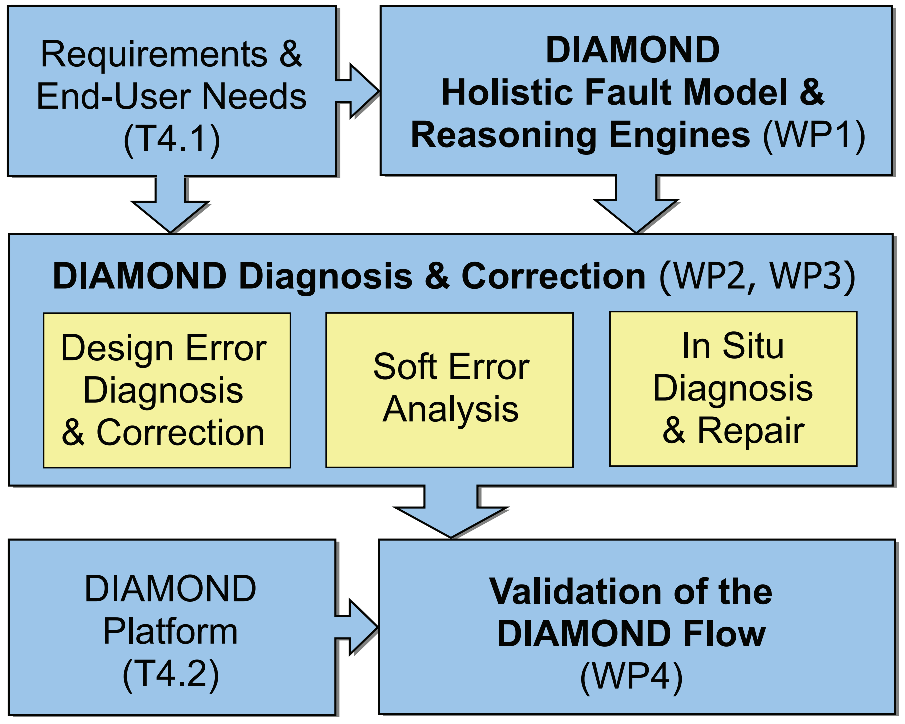

Figure 1: Overview of DIAMOND

This document deals with error localization in hardware designs that are described at transaction-level. A transaction-level model of a hardware design describes its functionality and basic structure on a high abstraction level. Implementation details like the exact timing of signals, bit-widths, and communication protocols are typically not included. On the transaction-level, designs are usually modeled by (simple) software programs. C, C++, or SystemC, which is an extension of C++ with a simulation kernel, are typical languages for such models. These languages come with a rich set of features like dynamic memory allocation, recursive function calls, and libraries of existing functions. This rich set of features allows the designer to quickly explore different design options and to validate basic functionality early in the development process. At the same time, it also renders automated reasoning about the design and its correctness difficult and challenging.

In order to automatically reason about the correctness of a transaction-level model, and in order to locate errors, some form of specification defining the desired behavior of the design has to be available. Such a specification may be given as a golden reference implementation, a set of assertions, or a test bench (see also [7]). Given a transaction-level model which violates its specification, error localization attempts to identify (sets of) components of the model which may be responsible for the violation. There are two additional objectives, which cannot both be maximized at the same time. On the one hand, the information about possible error locations should be precise, i.e., fine-grained and without too many false positives. On the other hand, the methods should be efficient and scalable.

In the DIAMOND project we developed various methods for transaction-level diagnosis, providing different trade-offs between these two main objectives. They can be classified into three major categories: formal methods, semi-formal methods, and dynamic methods. As illustrated in Fig. 2, these different kinds of methods have different characteristics. Formal methods transform the model into the domain of logic and use logic solving to compute diagnoses. They provide high reasoning power and, hence, the potential to produce precise diagnoses. On the other hand, they are computationally expensive. Dynamic methods execute the model with a given set of inputs. They are very scalable but notoriously incomplete. Semi-formal methods provide a compromise between these two extremes. They often execute the program but provide additional information about execution paths.

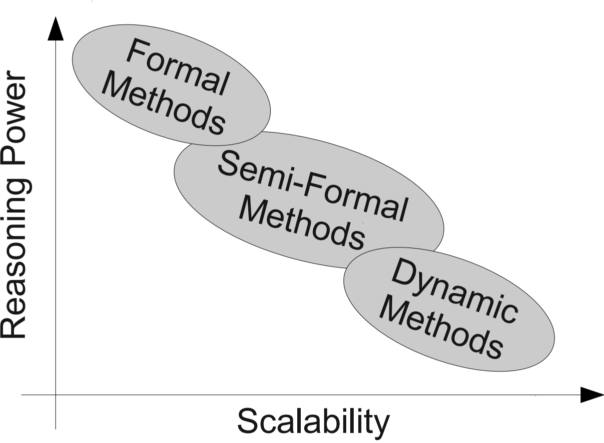

Figure 2: Different Characteristics of Different Methods

1.1 Overview of the Methods and the Document

Section 2 describes **formal methods** for error localization that have been developed within DIAMOND. A method to locate errors in unrealizable formal specifications is presented in Section 2.1. A specification is unrealizable if no system can implement it. Unrealizable specifications are a serious problem in all applications

in which complete or almost complete specifications are required. Section 2.2 discusses a method to diagnose inconsistencies between a hardware description at the register-transfer-level and its transaction-level specification via counterexamples.

Section 3 describes a **semi-formal method** to locate errors in transaction-level models. It is based on symbolic execution and model-based diagnosis, using a Satisfiability Modulo Theories (SMT) solver to find potential error locations.

Section 4 presents **dynamic methods** for diagnosis. Section 4.1 discusses a method to rank components of the transaction-level design regarding their suspiciousness. Dynamic program slicing is used to improve the accuracy of this method. Finally, Section 4.2 discusses a simulation-based algorithm for SystemC designs, which is based on program slicing as well.

Section 5 describes **integration and implementation** activities. Section 5.1 describes a multi-level fault-management architecture with an emphasis on diagnosis. A key feature of the architecture is that it enables fast fault detection with accurate fault localization. The architecture is designed in a hierarchical (tree-structured) way such that when a fault is detected at a leaf, it is propagated to the root. For fault detection, only the root has to be checked, while the hierarchical structure reduces the time spent for fault localization. Finally, Section 5.2 describes FoREnSiC, a prototype tool implementing various diagnosis methods for incorrect transaction-level models that are given as C programs. FoREnSiC has been developed from scratch within DIAMOND. A first release of the tool is planned for the end of 2011.

All activities presented in this document have been completed. The implementation of the techniques (cf. Section 5) is working, but it is likely to be improved and fine-tuned during the evaluation-phase of the project.

1.2 Link to the DoW

This deliverable summarizes activities in Task 2.1 *Transaction-Level Diagnosis*, which is part of WP2. The techniques presented address objective O2.1 *"Development of specification debugging and transaction-level error diagnosis techniques"* [6].

The Description of Work [6] reads:
"On the transaction level we will use a mix of formal, semi-formal, and dynamic techniques to combine efficiency with maximal discriminating power. We will also have to target faults in the specification itself."
Formal, semi-formal, and dynamic techniques are presented in the Sections 2, 3, and 4, respectively. A method for specification debugging is presented in Section 2.1.

Furthermore, the description of work states that we will deal with high-level models in C or in SystemC, and that we will make use of the reasoning engines developed in WP1. Most of the methods we developed are not specific for a certain

language. Our implementation in FoREnSiC operates on models written in C. The simulation-based method discussed in Section 4.2 diagnoses SystemC models. The different diagnosis techniques use various reasoning engines from WP1, e.g., model-based diagnosis engines, symbolic execution engines, and concolic execution engines.

Together with the deliverables D2.2b and D2.3b (to be submitted in March 2012), this deliverable contributes to milestone M2.2 *Prototypes of diagnosis tools implemented*. Prototype implementations are covered in Section 5.

2 Formal Diagnosis Methods

This section presents activities to use formal methods for error diagnosis. Formal methods are characterized by high reasoning power but also by a high amount of required computational resources (time and memory).

2.1 Activity: Diagnosis of Formal Specifications

Automatic error localization and correction of hardware designs requires a specification which defines the desired behavior of the circuit. Such a specification may be given in different forms (cf. Deliverable D1.2 [7]). One option is to use a formal specification. Creating a formal specification for a design is a challenging task and mistakes happen frequently. At the same time, debugging a formal specification is difficult, especially if no corresponding implementation is available. One reason is that the specification cannot be executed to track down the error, as one would do with an erroneous implementation.

This section presents an automated diagnosis method for unrealizable formal specifications of reactive systems. A reactive system is a system which continuously interacts with its environment. A specification for a reactive system is unrealizable if it is so restrictive that no system can implement it. Especially when trying to create a complete specification for a system, mistakes often lead to unrealizability [13, 14]. The presented debugging method considers the specification standalone, i.e., it does not require a corresponding implementation or higher-level specification. All it requires is a procedure to check a specification for realizability.

The diagnosis method for formal specifications has been published in the Haifa Verification Conference 2010 [14]. An extended version is accepted for publication in the International Journal on Software Tools for Technology Transfer (STTT). An implementation in the tool RATSY for so-called Generalized Reactivity(1) specifications [15] has been presented at CAV 2010 [2]. The reasoning engine underlying this debugging method (a model-based diagnosis engine) is described in Deliverable D1.3 [3]. The following sub-sections describe the debugging method on a relatively high abstraction level. For technical details, confer [14]. Two types of specifications are considered. The first one consists of a set of properties, the second one consists of environment assumptions and system guarantees. Finally, it is discussed how over-constrained signals can be identified.

Specifications Consisting of Properties

This type of specifications is given as a set P of properties for a system with inputs X and outputs Y. The specification requires the system to fulfill all properties. The specification is assumed to be unrealizable, i.e., so over-constrained that no system can implement it. The diagnostic challenge is to find out which parts of the specification may be responsible for unrealizability. Each individual property of the specification is considered to be one *component* of the specification. This makes sense because every property typically represents a relatively self-contained aspect of the system behavior. Therefore, the goal is to compute minimal sets of components that can be modified in such a way that the specification becomes realizable. These are exactly the properties that may be responsible for unrealizability.

A set of properties $\Delta \subseteq P$ can be modified in such a way that the specification becomes realizable if and only if the specification $P \setminus \Delta$ is realizable. This observation allows to compute diagnoses in the a naive way: For every possible subset Δ of P it can be checked if $P \setminus \Delta$ is realizable. If so, then Δ is a diagnosis, i.e., the properties in Δ may be responsible for unrealizability. Diagnoses can be computed more efficiently using model-based diagnosis. Deliverable D1.3 [3] describes the model-based diagnosis engine that is used as an underlying engine of this debugging method to compute diagnoses. As a side-product, this model-based diagnosis engine also produces conflicts. Conflicts are sets of components that cannot all be correct at the same time. These conflicts can be presented to the user as additional diagnostic information.

Specifications Consisting of Assumptions and Guarantees

The second type of specifications is of the form $A \rightarrow G$, where A is a set of environment assumptions, and G is a set of system guarantees. If the environment of the system fulfills all assumptions, the specification requires the system to fulfill all guarantees. In this setting, it does not make sense to compute assumptions which can be modified in such a way that the specification becomes realizable, because every assumption can be modified in such a way. In particular, modifying any assumption to false renders the specification realizable. Hence, the debugging method computes minimal sets of guarantees that can be weakened in such a way that the specification becomes realizable. This works essentially in the same way as described in the previous section.

Diagnosing Signals

The diagnostic question cannot only be formulated with respect to the different properties of the specification, but also regarding its signals. The goal is then to identify signals on which restrictions can be loosened to obtain a realizable specification. This problem can be solved by defining a special existential quantification

operator on signals, which removes all restrictions on the quantified signals. Removing restrictions on signals is similar to removing properties: it weakens the specification. The computation of diagnoses is similar as well. A set of signals is a diagnosis if removing restrictions on these signals using the existential quantification renders the specification realizable. Model-based diagnosis can again be applied to compute diagnoses efficiently. The two approaches of diagnosing signals and properties are orthogonal and can also be combined.

2.2 Activity: Diagnosis via RT-Level techniques

As described in the previous section, automated error localization and correction of hardware designs requires a specification which defines the behavior of the circuit. Often such a specification is given in a higher software-like language, such as e.g., C or C++. From such a specification, an implementation can be derived through a conversion to Verilog or VHDL. Since there are many differences between hardware and software, a correct conversion from software to hardware turns out to be a difficult task. For instance, in hardware designs computation is heavily parallelized. Furthermore, clocking schemes are added which synchronize the interaction between computational units that run in parallel. Thus, verifying the equivalence between software specifications and hardware implementations is a sophisticated but important field. After detecting discrepancies, diagnosing the problems is necessary.

This section describes an approach for verifying equivalence and diagnosing discrepancies between high-level specifications and corresponding RT-level implementations automatically. The reasoning engine underlying this approach is described in Deliverable D1.4 [4].

The presented approach considers ANSI-C specifications and RTL implementations. Both implementations in VHDL and Verilog are supported. In order to verify equivalence, the C program is mapped to a finite state machine (FSM) and then converted to the RT-level. The actual equivalence check is mainly driven by a satisfiability (SAT) solver. To perform an equivalence check, the corresponding primary inputs of both designs have to be determined and constrained to be stimulated by identical input values. Under these identical input values the corresponding outputs are determined and compared for identical functional behavior. This step involves heuristics to determine the correspondence between the untimed C model and the timed RTL model.

Since nowadays designs consist of billions of components, an equivalence check for whole designs is not possible due to capacity limitations of the SAT engine. Therefore, the conventional equivalence check is extended by a *cutpoint* detection which reduces the size of the verification instance significantly. Cutpoints represent parts within two designs which are assumed to be equivalent such that the verification procedure can be improved either by using the cutpoints as new start-

ing points to feed the designs with input values, or by reducing the verification problem by merging the functionally equivalent structures.

The diagnosis method running on top of this equivalence checking procedure determines mismatches between C specifications and RTL implementations and returns counterexamples for a detected mismatch. An implementation is part of FoREnSiC (see Section 5.2).

3 Semi-Formal Methods

This section describes diagnosis methods which are based on symbolic or concolic execution of the transaction-level model.

3.1 Activity: Diagnosis using Symbolic Execution

Symbolic execution is a program analysis technique which incorporates characteristics of dynamic methods and formal methods. Just like other dynamic methods, it executes the program. The difference is that symbolic execution uses symbols instead of concrete values for the inputs. Symbols are placeholders for any possible value. This allows for higher reasoning power when compared to purely dynamic methods, which can capture the program behavior for certain input values only. On the other hand, symbolic execution also produces predicates in some logic which express under which conditions a certain program path is activated. This is similar to other purely formal techniques which typically transform the program behavior directly into a formula in some logic. An essential difference is that symbolic execution analyzes the program path-by-path. This allows to trade reasoning power for efficiency quite easily: the more paths are analyzed, the more computational resources (time and memory) are required, but the higher the accuracy.

This section presents a diagnosis method for transaction-level models of hardware designs, which is based on symbolic execution. The symbolic execution engine itself is not described here. It is introduced as a reasoning engine in the deliverables D1.3 [3] and D1.4 [4]. This section contains a high-level summary of the diagnosis approach itself. The technical details have been published at the International Conference on Formal Methods in Computer Aided Design (FMCAD) [12]. An implementation is part of FoREnSiC (cf. Section 5.2).

Setting and Fault Model

We assume that the transaction-level model is given as a program in an imperative language, such as C. Assert statements in the code serve as specification. This also allows to use reference implementations as a specification: The program and the reference implementation can be executed with the same input and the outcome can be compared with user-defined assertions. The advantage of this approach is that the notion of equivalence can be defined by the user.

The diagnosis method should be able to identify components of the transaction-level model that may be faulty. Therefore, first of all a definition of what constitutes a component of the model is needed. The debugging method is able to handle the right-hand side (RHS) of every assignment as a potentially faulty component. The rest of the program is assumed to be correct. This fault model is able to handle all kinds of incorrect expressions, because every expression can be assigned to a temporary variable[1]. However, it cannot handle faults like, e.g., missing statements. This fault model was chosen because it allows for efficient program analysis.

Program Analysis

Before symbolic execution is carried out, it needs to be expressed that components may be faulty. This is done by textually replacing all assignments `LHS = RHS` by `LHS = cmp(c, RHS)`. Here, `cmp` is a special function indicating that the `RHS` may be faulty. The parameter `c` is a unique identifier of the component.

Next, symbolic execution is used to compute path conditions for the different execution paths of the program. Whenever the function `cmp` is called, a new symbol r is created. The symbol can take on any possible value. The new symbol is also associated with the symbolic value of `RHS`. This is the original value that would be produced by the component if the component was correct. It is denoted as $\text{Orig}(r)$ in the following. The component `c` that produced symbol r is denoted as $\text{CmpOf}(r)$.

The execution paths of the program can be divided into two sets: execution paths that end in an assertion violation, and execution paths that do not. For all paths that do not end in an assertion violation, the disjunction of the corresponding path conditions is computed. The resulting condition will be called $\pi(\bar{i}, \bar{r})$ in the following. It contains the input symbols $\bar{i}$ and the symbols $\bar{r}$ representing the unknown values returned by the calls to the function `cmp`. It evaluates to `true` if the program conforms to the specification when executed with the inputs $\bar{i}$ and when `cmp` returns the values $\bar{r}$.

A special form of symbolic execution is concolic execution. Here, the program is executed with symbolic and concrete inputs simultaneously. For the purpose of error localization, these two methods can be used interchangeably. Refer to the deliverables D1.3 [3] and D1.4 [4] for more details.

Computation of Diagnoses

The goal is to identify sets of components that can be modified in such a way that the program becomes correct. Such sets are called diagnoses. This definition

[1] Such a transformation is actually done in the implementation of this method.

makes sense because components that can be modified in such a way that the program becomes correct are exactly the components that may be responsible for the incorrectness.

A given set Δ of components is a diagnosis if the formula

$$\forall \bar{i} \,.\, \exists \bar{r} \,.\, \pi(\bar{i}, \bar{r}) \wedge \bigwedge_{\{r \in \bar{r} \mid \mathrm{CmpOf}(r) \notin \Delta\}} r = \mathrm{Orig}(r) \tag{1}$$

is satisfied. That is, for all inputs, there must exist some values that can be returned by the components such that the program behaves conforming to the specification. Components which are not part of the diagnosis must return the value that is returned by the original implementation of the component. If the above formula is satisfied, then this means that there exist some values that can be returned by the components in Δ such that the program becomes correct. This means that the components in Δ can, in principle, be modified in such a way that the program becomes correct.

Equation 1 contains a quantifier alternation which makes it hard to solve. Depending on the logic to express correctness and depending on the domains of the symbols, it may even be undecidable. Therefore, in practice, Equation 1 is not checked for all inputs but only for a given set of concrete input values. This can lead to false positives in diagnosis computation, but it increases the efficiency.

Equation 1 can be used to compute diagnoses in a naive way: every set of components can be check if it is a diagnosis. A more efficient algorithm, which is based on the computation of unsatisfiable cores and a hitting set tree, is presented in [12].

4 Dynamic Methods

This section explains methods to locate errors using dynamic methods. Dynamic methods have the potential to scale to large programs. However, their accuracy often cannot compete with formal or semi-formal methods.

4.1 Activity: Diagnosis via Dynamic Slicing

In this activity, dynamic slicing [16, 1] is applied in order to locate the causes of design errors in algorithmic descriptions. Program slicing is a technique for extracting portions of a program affecting a selected set of variables of interest. By focusing on the computation of only a few variables, the slicing process can be used to discard portions of the program which cannot influence these variables, thereby reducing the size of the program. The reduced program is called a slice. Slices represent a projection from the behavior of the initial program. This projection preserves the values of certain variables as seen at certain points in the program.

Figure 3 illustrates the concepts of static and dynamic slicing on a C code example. The leftmost column contains the statements of the example program. The next column shows the corresponding flowchart. The slice of the program is computed with respect to the variable `a` in the last line. That is, the slice answers the question which parts of the program may be responsible for the value of variable `a` at the end. This question can be answered statically, i.e., independent of a concrete execution. The third column in Figure 3 shows a static slice. It says that, under no circumstances, the statement `b=0` may be responsible for the value of `a` in the end. Slicing can also be done dynamically, i.e., using a concrete execution. Column 4 shows a concrete execution for `a=2`, `b=4` and `c=7`. The last column depicts the corresponding dynamic slice. It says that, in this execution, the initial value of `b` and the assignment to `b` are irrelevant for the value of `a` in the end.

This activity combines dynamic slicing with a ranking of the nodes [11] in the program flow-graph by calculating the suspiciousness score to each node n_i. Let a *failing slice* be a slice computed for an execution which results in erroneous values at the observable output of the system. All other slices are called *passing slices* in the following. Moreover, let F_i be the number of failing slices containing the flow-graph node n_i and let P_i be the number of passing slices containing n_i. Two options for calculating suspiciousness scores have been investigated. In the first one, the score of a node n_i is equal F_i, i.e., to the number of failing slices in which it occurs. The second option is to calculate the suspiciousness score of node n_i by $F_i/(P_i+F_i)$.

statements :	*P* :	*static slicing* :	*exec. statem.* :	*dynamic slicing* :
int a;	n_0	•	•	•
int b;	n_1	•	•	
int c;	n_2	•	•	•
if (c>0) {	n_3	•	•	•
b=0;	n_4		•	
c=3;	n_5	•	•	•
a=c+2;}	n_6	•	•	•
else {				
a=b-c;}	n_7	•		
out==a;	n_8	•	•	•

Figure 3: Example: A program and its slices

Our experiments show that the first option is more accurate in case of single design errors whereas the second option provides more robust results in the case of multiple simultaneous design errors. An implementation of this method has been done in the tool FoREnSiC (see Section 5.2).

4.2 Activity: Simulation-Based Debugging of SystemC

Simulation-based procedures are commonly used in different areas of application, both in software and hardware. They are a powerful approach to analyze large systems, since formal and semi-formal procedures with strong reasoning capabilities have potentially large resource requirements regarding run time and memory. SystemC increasingly receives attention since SystemC allows for specifying hardware designs in a more abstract way than standard hardware description languages do. SystemC is based on C++, extending the language with class libraries and a simulation kernel.

This section presents a simulation-based debugging algorithm based on program slicing. The objective of the procedure is to reduce the debugging effort by focusing the attention of the designer on a subset of program statements which are expected to contain faulty code.

Whenever one obtains erroneous output values from simulation, for a given SystemC design, faulty computation steps must have occurred. The resulting simulation traces are expected to contain faulty code so that they focus the attention on a subset of the design to be debugged.

The reasoning engine underlying this diagnosis method is described in Deliverable D1.4 [4]. The technical details of this work have been presented at the *Forum on Specification & Design Languages* [8]. Furthermore, the work has been accepted for inclusion in a book on the best papers of the conference published in the series *Lecture Notes in Computer Science*.

5 Integration and Implementation

This section covers integration and implementation activities to enable error diagnosis in practice. It describes a fault-management infrastructure for multi-processor systems. It also briefly describes the tool FoREnSiC implementing many of the techniques described above.

5.1 Activity: Fault Management Infrastructure

This section presents an infrastructure to enable diagnosis in a multi-processor system-on-chip (MPSOC). Parts of the work have been published in [10].

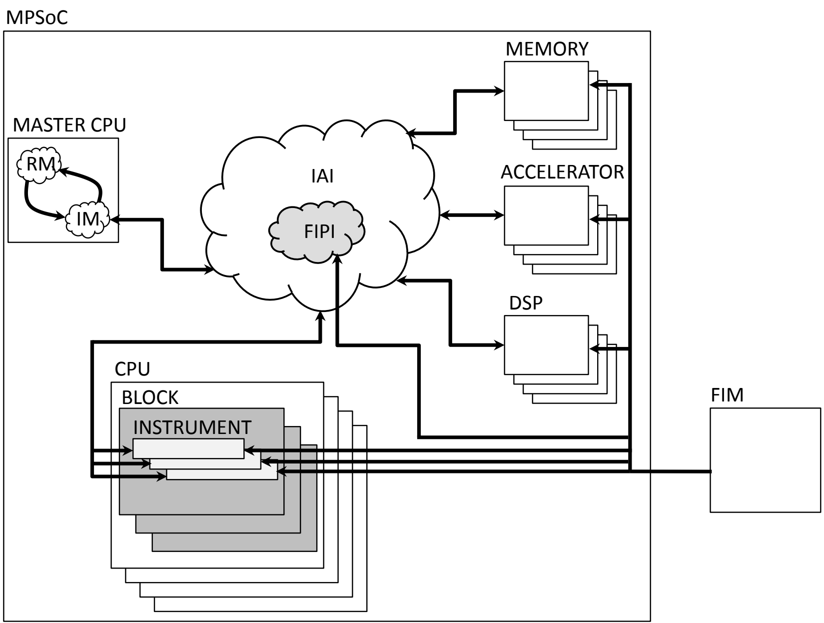

Figure 4: An MPSOC with CPUs, DSPs, accelerators, and memories

A typical MPSOC is shown in Fig. 4. The MPSOC consists of a set of components such as CPUs, DSPs, accelerators and memories. Each component consists of one or more blocks. Fig. 5 shows the ALU block and the control (CTRL) block of the CPU in more detail. The ALU block consists of a scan-chain, which is primarily used for manufacturing tests. The CTRL block consists of a program counter (PC)

and a register to enable emulation of program execution. There is one master CPU responsible for assigning jobs to the components. For experimentation, the Fault Injection Manager (FIM) injects soft and hard faults into the MPSOC.

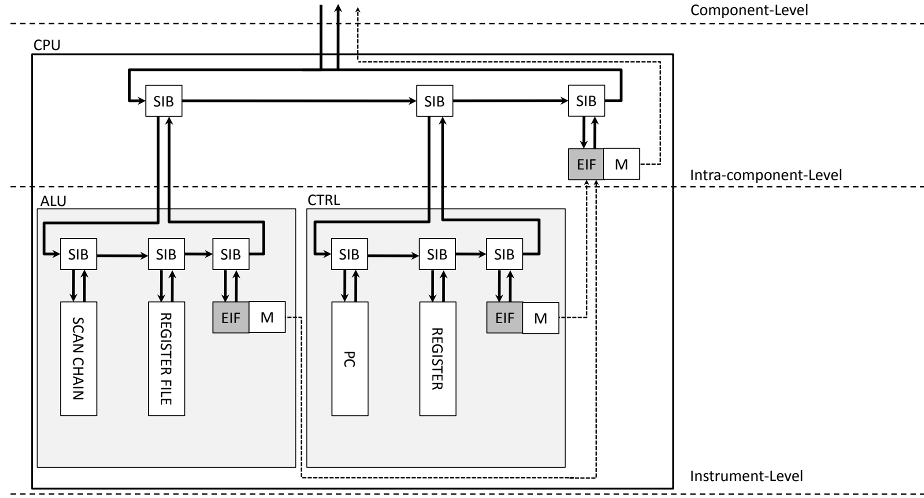

Figure 5: A CPU with an ALU block and a control (CTRL) block

The master CPU, responsible for assigning jobs to components, needs to receive a fault status from each component such that the assignment of jobs to components can be optimized. The requirements on sending the fault information is that it should have a minimal impact on the functionality.

Each block is assumed to have a fault detection mechanism. The infrastructure connecting blocks and components with the master CPU is constructed to enable fast notification if an error occurs somewhere in the MPSOC and precise diagnosis to accurately pinpoint the fault location. The infrastructure consists of a Failure Indication and Propagation Infrastructure (FIPI), shown as dotted lines in Fig. 4. When an error occurs in a block, for example in a register file of the CPU, the error indication flag (EIF) of the register is set. The error indication is also propagated to the component-level, the component type-level, and the system-level. The EIF-bit at the system-level indicates that an error has occurred.

The Instrument Access Infrastructure (IAI), which follows the proposal for IEEE standard P1687 to standardize the access to on-chip instrumentation, is designed such that segment insertion bits (SIBs) are added to allow flexible access of instruments, including EIF-flags. Typically, the system-level EIF is constantly polled to determine if an error is present in the system. If an error is present, the IAI can be reconfigured such that the error is localized level-by-level. This is illustrated in Fig. 6.

At system-level, the Resource Manager (RM) is responsible for collecting fault statuses from instruments, conducting fault handling tasks, and scheduling jobs based on the fault statuses. The RM gives commands to the Instrument Manager (IM) which acts as the interface between the RM and the instruments.

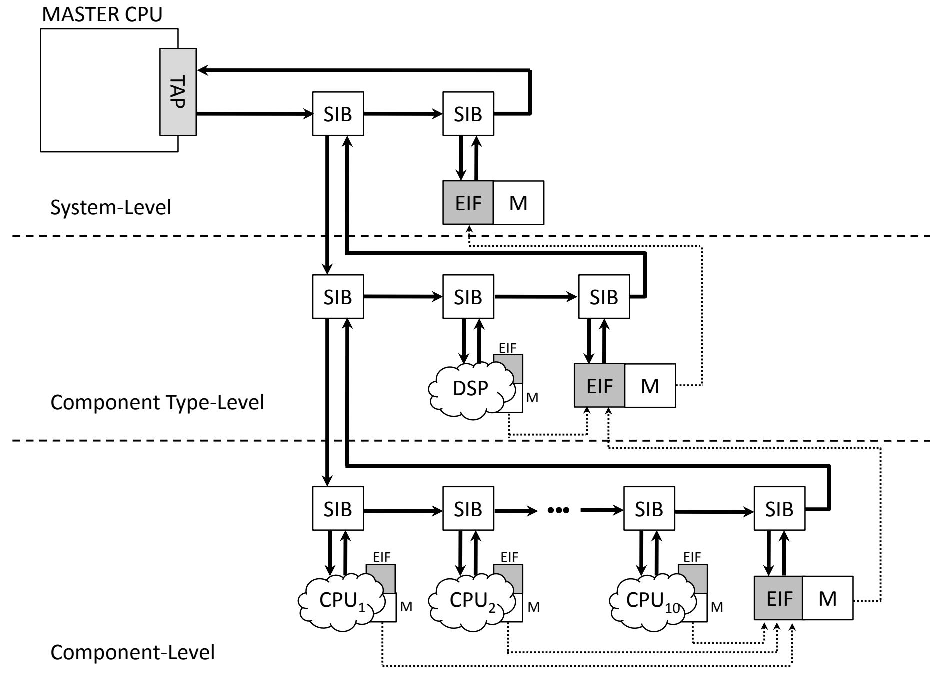

Figure 6: The diagnostic infrastructure in detail

5.2 Activity: FoREnSiC

FoREnSiC is a tool to automate error localization and correction for transaction-level models which are given as C programs. It has been developed from scratch within the frames of the DIAMOND project. A first release as an open-source tool is planned for the end of 2011. The hope is that, with this release, we draw additional attention to the research done in DIAMOND. Moreover, sharing the implementations of various debugging methods with the community is supposed to stimulate further research in this interesting field.

FoREnSiC stands for "**Fo**rmal **R**epair **En**gine for **Si**mple **C**", but actually the title is not fully accurate any more. FoREnSiC has grown to be more. First of all, it is not *one* engine but a bundle of several engines implementing various different debugging techniques. Second, it is not purely formal. The techniques implemented in FoREnSiC range from simulation-based methods to semi-formal and formal ones. Third, it does not only address repair of programs but also error detection and localization. Finally, FoREnSiC cannot only debug C programs but also hardware, with a C program functioning as a specification for this hardware.

The Architecture

Figure 7 depicts the architecture of FoREnSiC in a simplified form. A (potentially faulty) C program is the main input for the tool. The front-end of FoREnSiC parses this C program and produces an internal model of the program. FoREnSiC has sev-

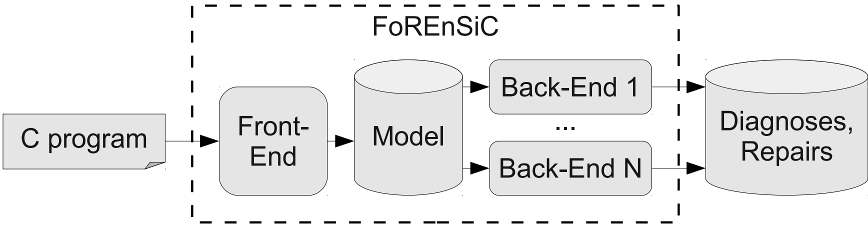

Figure 7: The architecture of FoREnSiC.

eral back-ends operating on the model of the program. They implement different error localization and correction methods. The user selects the back-end which is best suitable for her problem.

Certain back-ends require additional inputs. This may include test vectors with expected outputs or other forms of specifications, hardware implementations to check equivalence with, or the string representation of the C program. These additional needs are not drawn in the figure for simplicity. The value of FoREnSiC must not only be seen in the back-ends but also in the infrastructure it provides. New debugging methods can be added easily in form of new back-ends. The following sections explain the different parts of the architecture and the different back-ends in more detail.

The Front-End

The front-end is based on the GNU Compiler Collection (GCC) plug-in API version 4.5.0 and hooks into the GIMPLE pass [9]. The front-end processes the input program as a tree in low level GIMPLE SSA (Static Single Assignment) form. The compiler is invoked without the linking stage, so it is possible to process programs without a main function or with undefined functions.

The front-end should work on 32-bit and 64-bit platforms. The input program has to be a correct ANSI C (C90) program, including the GNU dialect of ISO C90. If the compiler finds an error, no model is built. Warnings are ignored by the plug-in. Currently only one input file is supported.

Besides the GCC-based front-end, FoREnSiC also contains a more primitive front-end which does not handle the full ANSI C language. This primitive front-end was important in the early phases of the tool development. It allowed the development process to start with the implementation of the back-ends already before the more robust GCC-based front-end was ready. Meanwhile, this primitive front-end is obsolete.

The Internal Model

This section explains briefly how a program is represented inside FoREnSiC. The front-end transforms the program into this internal representation. The back-ends then use these data structures for automated debugging.

The model of the program contains information about global variables and their types. Moreover, for every function, it stores its parameters and local variables with their type. The actual body of a function is represented as a flowgraph. Every node represents a statement, edges between the nodes express the control-flow. The flowgraph distinguishes nodes representing operations and nodes representing conditions. The former have only one successor, the latter have two. For every statement, its string representation as well as information about the location of the statement in the input file is stored. This allows to back-annotate results and to present information about error locations and corrections to the user. For every statement, the model contains also an abstract syntax tree. This is a tree-like decomposition of the statement into its operators and operands. This abstract syntax tree allows for easier analysis and evaluation of statements. The representation of data types distinguishes primitive types and compound types such as structs or unions. Every type can also occur as pointer or array of this type. For non-primitive types the location of their definition is stored as well.

The Equivalence Checking Back-End

The equivalence checking back-end implements the equivalence checking method explained in Section 2.2 together with its diagnosis capabilities. In contrast to the other back-ends of FoREnSiC, the C program serves as a specification (for a hardware design) and not as the debugging subject here. The hardware design may be given in Verilog or VHDL at the register transfer level. In the C program, this back-end does not support compound types such as structs and unions, as well as pointers. These are reasonable assumptions for programs which model hardware designs. The user can place special annotation in the C program to indicate that certain variables are inputs of the design. The inputs and outputs of the C specification are matched with that of the hardware design via their names.

The Symbolic Back-End

The symbolic back-end implements the diagnosis method introduced in Section 3.1. This section focuses on a few implementation aspects of the diagnosis engine.

The diagnosis engine can be operated in two different modes, a conservative mode and a progressive one. The two modes differ in how they treat incompleteness in program analysis. In the conservative mode, the program is only considered correct if a termination of the program can be enforced without an assertion violation. In the progressive mode, the program is deemed correct if all known assertion

violations can be avoided. Note that this has the effect that also endless loops are considered as correct behavior. The conservative method may miss diagnoses, the progressive mode may find too many diagnoses. Both have their merits. Diagnoses are computed in order or increasing cardinality. Usually, diagnoses with lower cardinality are considered to be more likely and helpful. This means that, when aborting the computation before all diagnoses have been computed, only less likely diagnoses are missed. The diagnosis engine provides a number of parameters with which its behavior can be fine-tuned and adjusted for a particular problem instance.

The Simulation-Based Back-End

This back-end implements the debugging method described in Section 4.1. As an input, it takes a (potentially incorrect) C program and a set of input vectors with corresponding expected output vectors. An alternative to the expected outputs is to feed the back-end with a reference implementation. In this case, the expected outputs are computed by simulating the reference implementation with the provided inputs first. Special annotations can be used to define which program variables are inputs or outputs of the program.

Error localization proceeds as follows. First of all, the program is instrumented with calls to special functions. These functions store which statements have been executed. Furthermore, they keep track of the variables that have been read or written in each statement during the execution of the program. This information is needed in order to compute a slice of the execution. The instrumented version of the program is now compiled using a standard compiler and executed repeatedly using the provided test vectors. An execution is classified as *failed* if the produced output values do not match the expected ones, or if an assertion has been violated during the execution. The execution is classified as *passed* otherwise. The slices are computed and the rank of the different nodes of the flowgraph are calculated as explained in Section 4.1. The nodes with the highest ranks are presented to the user as most likely fault candidates.

6 Summary

This document summarizes the main achievements in Task T2.1 *Transaction-Level Diagnosis*, which is part of WP2 in the DIAMOND project. This is the final deliverable for this task. Together with the deliverables D2.2b and D2.3b, it contributes to milestone M2.2 *Prototypes of diagnosis tools implemented*.

The presented transaction-level diagnosis techniques include formal, semi-formal, and dynamic techniques. All categories of techniques have their strengths and weaknesses regarding reasoning power and scalability. The formal techniques include methods to diagnose over-constrained formal specification and to diagnose inconsistencies between hardware descriptions and their transaction-level specification. The semi-formal methods locate errors in transaction-level models of the hardware, based on symbolic or concolic execution of the model. The dynamic techniques use program slicing to obtain more accurate information about potential error locations. Finally, a fault management infrastructure for multi-processor system-on-chips is presented, and an implementation of various error localization techniques in the tool FoREnSiC is discussed.

Concerning Task 2.1, the Description of Work [6] requires to develop specification debugging techniques, to deal with high-level models in C or in SystemC, to develop formal, semi-formal, and dynamic techniques, and to make use of the reasoning engines developed in WP1. This deliverable explains how these objectives haven been addressed.

7 References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Programming Language Design and Implementation*, pages 246–256, 1990.

[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] 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.

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

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

[8] 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.

[9] Inc. Free Software Foundation. *GNU Compiler Collection (GCC) Internals*, 2010.

[10] F. Ghani Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson. Access time analysis for IEEE P1687. *IEEE Trans. on Comp.*, PP(99):1, 2011.

[11] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *International Conference on Automated Software Engineering (ASE'2005)*, pages 273–282. ACM, 2005.

[12] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Int'l Conference on Formal Methods in Computer Aided Design*. IEEE, 2011. To appear.

[13] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *Int'l Conference on Formal Methods in Computer Aided Design*, pages 152–159, 2009.

[14] R. Könighofer, G. Hofferek, and R. Bloem. Debugging unrealizable specifications using model-based diagnosis. In *Proceedings of the Haifa Verification Conference*, volume 6504 of *LNCS*. Springer-Verlag, 2010.

[15] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.

[16] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.