Final Report for DFG project No. FE797/15-1

Title: Methodology, Algorithms, and Framework for Hardware Design Understanding

Prof. Dr.-Ing. Görschwin Fey, TU Hamburg, Germany

1 General information

DFG reference number:	FE 797
Project number:	FE 797/15-1
Project title:	Methodology, Algorithms, and Framework for Hardware Design Understanding
Name of the applicant:	Görschwin Fey
Official address:	Technische Universität Hamburg, Institut für Eingebettete Systeme, Am Schwarzenberg-Campus 1, 21073 Hamburg
Reporting period:	June'21–August'24 (effort: 24 person months)

2 Summary

English: Understanding a given digital system design one is unfamiliar with is at least as hard as implementing it. Particularly, this holds for systems designed in large projects where no single person knows all the details, legacy components with poor or outdated documentation are reused, and team members change regularly. Designers then try to understand the inner logic of the design using various sources – informal discussions with team members, textual information like documents for requirements and specifications, and review as well as execution of source code and test cases.

This project created a tool to support designers in understanding the source code of a system design at the Register Transfer Level (RTL). The main focus was on techniques and algorithms known from security analysis and their application for design understanding, investigating the first steps towards advanced user interfaces, providing benchmarking cases, and open-sourcing the resulting implementation. The tool DuRTL is now available in its first release: `https://github.com/TUHH-IES/DuRTL`.

Moreover, the recently drastically increased capabilities of Large Language Models (LLMs) directly relate to the theme of the project, so initial steps toward assessing their capabilities have been made.

Deutsch: Einen Entwurf eines digitalen Systems, mit dem man nicht vertraut ist, zu verstehen, ist mindestens so schwierig wie dessen Implementierung. Dies gilt insbesondere für Systeme, die in großen Projekten entwickelt werden, in denen keine einzelne Person alle Details kennt, existierende Komponenten mit schlechter oder veralteter Dokumentation wiederverwendet werden und die Teammitglieder regelmäßig wechseln. Designer versuchen dann, die innere Logik des Entwurfs zu verstehen, indem sie verschiedene Quellen nutzen – informelle Diskussionen mit Teammitgliedern, textuelle Informationen wie Dokumente für Anforderungen und Spezifikationen sowie die Überprüfung und Ausführung von Quellcode und Testfällen.

In diesem Projekt wurde ein Software-Werkzeug entwickelt, das Designerinnen beim Verstehen des Quellcodes eines Systementwurfs auf der Register-Transfer-Ebene (RTL) unterstützt. Der Schwerpunkt lag auf Techniken und Algorithmen, die aus der Sicherheitsanalyse bekannt sind, und deren Anwendung für das Verständnis des Entwurfs, die Untersuchung der ersten Schritte in Richtung fortgeschrittener Benutzerschnittstellen, die Bereitstellung von Benchmarking-Fällen und das Open-Sourcing der resultierenden Implementierung. Das Werkzeug DuRTL ist nun in seiner ersten Version verfügbar: `https://github.com/TUHH-IES/DuRTL`.

Darüber hinaus stehen die kürzlich drastisch gestiegenen Fähigkeiten von Large Language Models (LLMs) in direktem Zusammenhang mit dem Thema des Projekts, so dass erste Schritte zur Bewertung ihrer Fähigkeiten unternommen wurden.

3 Progress report

Design understanding is a time-consuming task required for debugging, validation, verification, design extensions, and other typical tasks a designer may carry out. The core ideas of this project are twofold:

- Reusing existing knowledge available through testbenches that are provided with virtually any design.

  In contrast to other documentation, a testbench is necessarily up to date with respect to the most recent design changes whenever a typical methodology based on continuous integration and source code versioning is used. At the same time, the testbench captures typical use cases for a design.

- Adjusting Information Flow Tracking (IFT), known from security analysis, for an understanding task.

  IFT is an automated technique originally aiming at analyzing the flow of information of well-defined confidential data items. While these confidential data items are known in security analysis, the task is more open in design understanding. However, automated extraction of typical information flows helps to understand the working principles of a design. Automated extraction is the challenge.

In Section 3.1, we briefly review the objectives and background of IFT together with some specific adjustments made for design understanding and design decisions taken. In Section 3.2, we discuss our results: the open-source tool DuRTL and extensions explored beyond the main flow of the project. We review the alignment with the original concept of the project in Section 3.3, quality enhancing measures in Section 3.4, and handling of research data in Section 3.5.

3.1 Background and Objectives

We briefly review the main objectives considered in the project proposal. Then, we introduce IFT to have the report self-contained and to mention extensions and decisions related to design understanding.

3.1.1 Objectives

The main objective was to devise and evaluate a holistic methodology to support design understanding. The project was planned for four funding years, and we report on results from the first two years. Within these first two years, the goals were to formalize the methodology, set up benchmarking examples, study use cases, and create the necessary implementation for the evaluation.

3.1.2 Fully Automated Information Flow Tracking for Understanding

IFT relies on a well-defined concept of information flow. In this context, information flow refers to the movement and transformation of data within a system, encompassing the physical transmission of data from one location to another and how it is processed, stored, and accessed throughout the system's components. Conceptually, information flow is understood as follows:

> *Information flows from a signal a to a signal b under a specific assignment if and only if changing the value of a causes a change in the value of b.*

A design is a netlist composed of gates connected by signals. Technically, our netlists are given at the word level, i.e., complex gates like adders or multipliers may be used. These netlists are also hierarchically organized, i.e., some modules may be instantiated multiple times within other modules. An instance of a module can be considered as a special gate type from the perspective of the parent module.

We extend the previous approaches [4, 5] by keeping track of time, considering an arbitrary number of tags instead of having single bits tracking information flow and considering bit-vector level information flow.

At the core of the IFT technique is the concept of *tags*, which are metadata associated with various pieces of data or signals within a circuit or software system. These tags help identify the source and type of data, enabling the tracking of its flow throughout the system over time. IFT starts when a *tag* is injected into a specific signal s at a specific time t into the circuit during simulation. The *tag* is univocally associated with a signal and identifies a flow of information during the traversal of the circuit. As signals or data packets move through different system components, their tags are updated depending on the operations performed on them. In practice, a signal *s* at time *t* has a tag set $\mathsf{T}_{s,t} = \{\tau_0, \tau_1, \ldots, \tau_n\}$, where $n \in \mathbb{N}$ is the total number of time steps in the simulation multiplied by the number of tag injection points.

Table 1: Tag sets for an AND-gate.

g	h	T_f
$\neq 0$	$\neq 0$	$T_g \cup T_h$
$\neq 0$	0	T_h
0	$\neq 0$	T_g
0	0	$T_g \cap T_h$

Table 2: Tag sets for D-flip-flop.

clk	$T_{f,t}$
rising edge	$T_{D,t}$
other	$T_{D,t-1}$

Precision The precision and the complexity of IFT depend on how the propagation of tags between signals is decided. The definition for precise IFT is based on [4]. Tag propagation is called precise when a tag is associated with an output signal *if and only if* the tag was associated with an input whose change affects the output. This decision is usually made under specific assignments to all the inputs of a design. More formally, we consider a design d, with inputs $I = (i_1, \ldots, i_n)$ and outputs $O = (o_1, \ldots, o_m)$. Let $o^t(I_1, \ldots, I_t)$ be a function to determine the value of output o of a design at time step t under an input sequence over t time steps. Let $A_t = (a_{1,t}, \ldots, a_{n,t})$ denote a fixed assignment to the inputs $I_t = (i_{1,t} \ldots i_{n,t})$ at time t. Then, the following equation defines the propagation of tags from inputs to outputs of that design:

$$\mathsf{T}_{o,t} = \bigcup_{j=1}^{t} \bigcup_{k=1}^{n} \exists a'_{k,j} o^t(A_1, \ldots, (a_{1,j}, \ldots, a_{k,j}, \ldots, a_{n,j}), \ldots, A_t) \neq o^t(A_1, \ldots, (a_{1,j}, \ldots, a'_{k,j}, \ldots, a_{n,j}), \ldots, A_t) \; ?\mathsf{T}_{i_k,j} : \emptyset \quad (1)$$

where $a'_{k,j}$ represents the change in the input i_k at time j.

Instead of input and output signals, any other signal and its transitive predecessors may be considered to determine whether information flows from the predecessor to the signal. The definition may be generalized to sets of predecessors and internal signals that jointly induce propagation flow, but here, we stick to the simple view of a single signal.

Such a precise approach is, in general, computationally costly as a global view, considering possible reconvergences of signals and resulting functional dependencies that must be taken into account under all possible value changes. Thus, for efficiency, approximate analysis is often used. An over-approximation may add information flows not actually existing under Eq. (1). Using only a subset of all stimuli for the analysis generally causes an under-approximation. However, by focusing on usage scenarios defined by the testbench, this is the desired behavior.

Propagation Rules The propagation rules for IFT compute an over-approximation of information flow with respect to the stimuli given by the testbench. Instead of computing Eq. (1) globally on the circuit, these propagation rules consider a single (potentially complex) gate locally.

AND-Gate Table 1 shows the propagation rules of an AND-gate that computes the bit-wise AND over two bit-vectors g and h. For any combinatorial logic gate, the time step is the same at the inputs and outputs of the gate in the circuit, so in $T_{f,t}$, $T_{g,t}$, $T_{h,t}$, the time t is omitted.

If either input $g \neq 0$ or $h \neq 0$, the tag set of the other input is propagated. Thus, if both inputs are non-zero, the tag sets of both inputs are joined and propagated. For being conservative w.r.t Eq 1, the case $g = 0 \;\&\&\; h = 0$ propagates the intersection of both input tag sets. Both inputs of a gate may be influenced by the same input of the design, where a change in the design input would cause both gate inputs and, with that, the gate output to change, which can cause an output of the design to change as well.

The following equation implements the rules from Table 1 where $d = (a \; ? \; b \; : \; c)$ uses the ternary operator $?$ to denote 'if a then b else c':

$$\mathsf{T}_f = (g \; ? \; \mathsf{T}_h : \emptyset) \; \cup \; (h \; ? \; \mathsf{T}_g : \emptyset) \; \cup \; (\mathsf{T}_g \; \cap \; \mathsf{T}_h) \quad (2)$$

D-type Flipflop The rules for the tag propagation of D-type flipflops are shown in Table 2, where $T_{D,t}$ represents the tag set of the input D of the flipflop at time t.

The tag propagation for D-type flipflops is working just as the flipflop itself. The tag at the input is propagated at the same time the input value is propagated at the rising or falling clock edge, depending on how the flipflop is implemented. In between, the tag is held the same as the actual flip-flop value.

Multiplexer We consider a multiplexer (MUX) $f = s\ ?\ g : h$, so g and h are two bit-vectors for the data inputs, and s is the select signal which determines which input is passed to the output f of the multiplexer. The multiplexer is a combinatorial gate, so the time is omitted in the propagation rule:

$$\mathsf{T}_f = (s\ ?\ \mathsf{T}_g\ :\ \mathsf{T}_h)\ \cup\ (g \neq h\ ?\ \mathsf{T}_s : \emptyset)\ \cup (\mathsf{T}_h \cap \mathsf{T}_s)\ \cup\ (\mathsf{T}_g \cap \mathsf{T}_s) \tag{3}$$

The last two terms for the multiplexer take a more global view into account. If one of the signals g or h shares an input connection with the select signal s, tags common to both signals are propagated. In addition to the cells shown here, the rules for the propagation logic have been extended to other RTL cells like adders, registers, etc., with the details being omitted for brevity. All definitions in regards to the propagation logic do not only hold for 1-bit signals but also bit-vectors

3.2 Project-specific Results and Findings

Our open-source tool formalizes and canalizes the methodology proposed for design understanding. So, we describe our open-source tool at first, including some design decisions taken. Then, we briefly review the results for an evaluation of two use cases in Section 3.2.3. Section 3.2.4 discusses extensions and additional results beyond this main project focus.

3.2.1 DuRTL: A Framework for Hardware Information Flow Tracking

As part of this project, the design information flow analysis tool DuRTL – **D**esign **u**nderstanding for **R**egister **T**ransfer **L**evel – was developed [11],[SMF24]. Currently, DuRTL includes a tag-based hardware IFT approach. The tool is implemented in C++ and parses Verilog hardware netlists. Using the included IFT approach, DuRTL can identify the information flow of a design and, depending on the following analysis steps, use this information to highlight important parts of the design. Previous approaches to IFT were focused on hardware security. DuRTL instead focuses on understanding and comprehending unknown hardware designs written in Verilog by analyzing all information flows of the design.

The main requirements for DuRTL were: 1. dynamic analysis of hardware designs including storage of simulation data, 2. extendability, 3. scalability, and 4. support for Verilog hardware designs.

3.2.2 Architecture of DuRTL and Methodology for Design Understanding

In this section, we give an overview of the functionality of DuRTL using a simple example circuit before diving into the implementation details. The tool flow of DuRTL is shown in Figure 1. This also captures the methodology for design understanding considered here. The tool reads a hardware design and creates an internal representation. The internal representation is used for analysis and simulation of the design. The tool can export the design back to Verilog and export testbenches for IFT. The simulation results include tagging information that can be analyzed using the internal representation of the design.

The tool flow consists of the following steps:

- Design Parsing ① ②
- Design Export ③
- Testbench Export ④ ⑤ ⑥
- Simulation ⑦
- Analysis ⑧

We illustrate the tool flow using the example circuit shown in Listing 1. The circuit is a simple AND gate in a module that is instantiated in another module.

The internal representation consists of three parts: the design class structure, the module graphs, and the design instantiation graph. Figure 2 shows the instantiation graph of the example circuit that is part of the internal representation. The design consists of two modules. The top module `B` (light nodes) instantiates the module `A` (dark nodes). The module `A` is an AND gate that takes two inputs `x` and `y` and outputs the result `z`.

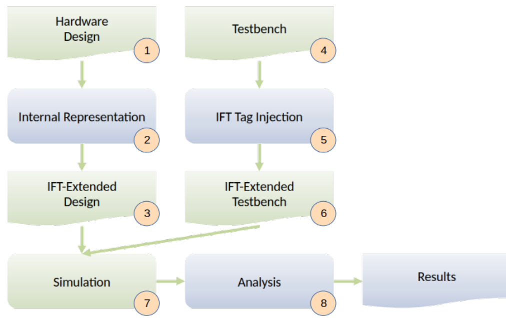
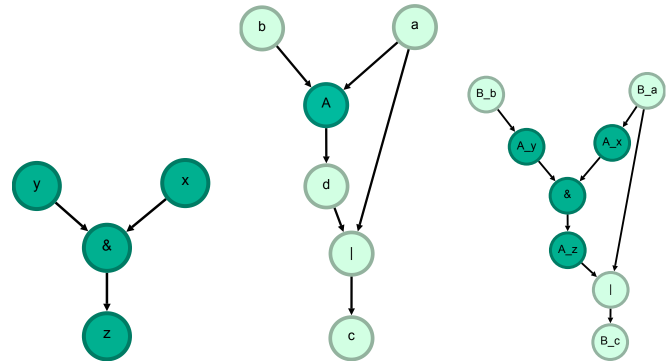


Figure 1: DuRTL tool flow. The circled numbers represent the steps of the tool flow.



(a) Module graph for 'A' (b) Module graph for 'B' (c) Instantiation graph

Figure 2: Internal representation graphs for the example circuit in Figure 1.

The instantiation graph of the design is a directed graph where the nodes represent the ports and cells of the design, and the edges represent the connections between the ports and cells.

The design is then exported with added IFT logic. Listing 2 shows example code of the exported module `A` with added IFT logic. The IFT logic is added to the module by adding tags for the inputs and outputs of the module and adding a logic equation that decides which tags are propagated. Listing 2 also shows the IFT logic equation that was added for the assignment of the output `z` in line 4 of the original code in Listing 1. Analogously, Listing 2 shows the added tag ports for the module `A` that were added for the ports shown in lines 1-3 of the original code in Listing 1. The tags are added to the Verilog code by extending the input and output ports of the module with additional tag ports as shown in Listing 2. The tags are then used in the propagation logic to decide which tags are propagated, thus tracking the information flow of the module.

Listing 1: Example circuit in Verilog.

```
module A(x, y, z);
  input x, y;
  output z;
  assign z = x & y;
endmodule

module B(a, b, c);
  input a, b;
  output c;
  wire d;
  A modins(.x(a), .y(b), .z(d));
  assign c = a | d;
endmodule
```

Listing 2: Code excerpt of the exported module A with added IFT logic. Example of tag ports being exported in Verilog.

```
module A(x, x_t, y, y_t, z, z_t);
  input  x;
  input  [31:0]  x_t;

  ⋮

  assign z = x & y;
  assign z_t =
    (^x === 1'yx ? 0 : (x > 0 ? y_t : 0)) |
    (^y === 1'yx ? 0 : (y > 0 ? x_t : 0)) |
    (x_t & y_t);
endmodule
```

In the following, the implementation of the different stages of the tool flow are briefly described.

3.2.2.1 Design Parsing DuRTL does not read Verilog code directly. Instead, the third-party tool YOSYS[1] is used to parse the Verilog code and export the design as a netlist in JSON format (1). DuRTL utilizes a third-party JSON parser[2] to parse the JSON file. Then, DuRTL parses the netlist contained in the JSON file and creates the internal representation of the design (2).

3.2.2.2 Internal Representation The tool uses the Boost graph library[3] to represent the internal structure of the design as a graph (2). The internal representation consists of two parts: the design class and the design instance graph. The design class is based on the structure of the netlist exported by YOSYS. The JSON netlist exported by YOSYS contains every module of the design. Every module is further subdivided into ports, cells, and nets.

[1] Yosys Open Synthesis Suite, `https://github.com/YosysHQ/yosys`
[2] JSON for Modern C++, `https://github.com/nlohmann/json`
[3] Boost C++ Libraries, `https://www.boost.org`

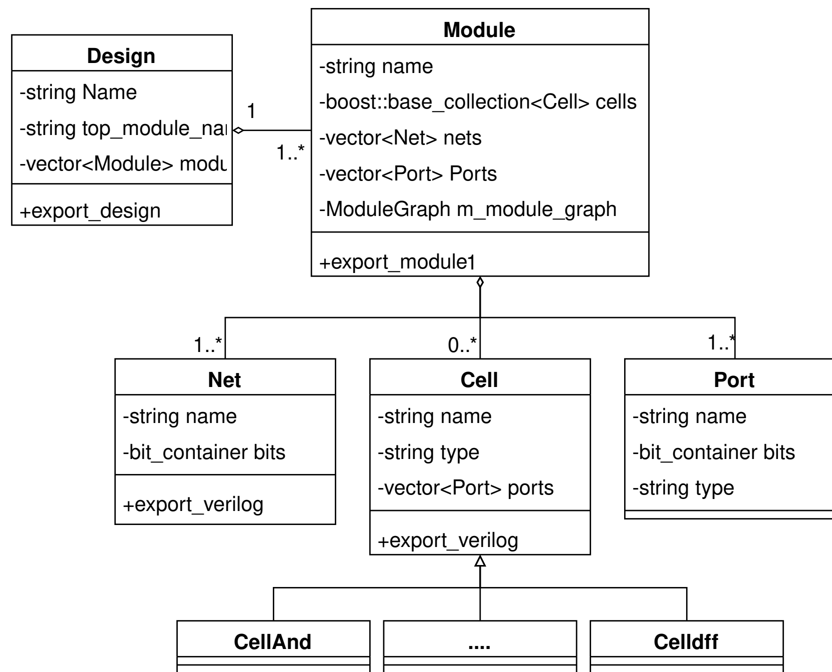


Figure 3: Class diagram for the internal representation of the design.

- `Ports` are all input/output/inout ports of the module
- `Cells` contain any word-level cells (e.g., and, or, mux, dff) of the module
- `Nets` contain all word-level nets that connect cells and ports

The design is stored in a `Design` object, which contains all the modules of the design. Each module is stored in a `Module` object, which contains the cells and nets of the module. A `Design` is the topmost class that holds all the `Module` objects. A `Module` holds `Cell` and `Net` objects. A `Cell` is a base class for all cell types. Every cell type has a corresponding class that inherits from `Cell`, e.g. `CellAnd` and `Celldff`. The cell classes contain the logic for exporting the cells to Verilog.

Design Choice 1 *Avoid Explicit Dynamic Memory Allocation*
We use `boost::base_collection<Cell>` to store objects of classes that inherit from `Cell`. This lets us store those objects in a single collection, avoiding the need to perform explicit dynamic allocation.

The internal representation contains two types of graphs. The module graph and the design instantiation graph. The module graph represents the connections between the elements of a module. The design instantiation graph represents the connections between the elements of the design.

Design Choice 2 *Internal Representation as Graphs*
A graph representation of the design is a very flexible way to represent the connections between the elements of a hardware design. The Boost graph library is very powerful and provides a lot of functionality for working with graphs.

3.2.2.3 Module Graphs Each module has an internal graph representation `ModuleGraph`. The `ModuleGraph` is a `boost::adjacency_list<>`, which simplifies traversing the module elements and their connections.

The module graph uses the Boost graph library to represent a hardware design as a graph. Every port, cell, and net in the design is a vertex of the graph. The edges of the graph represent the connections between elements.

3.2.2.4 Design Instantiation Graph A `DesignInstance` contains a flattened graph representation of the design using `boost::adjacency_list<>`. The design instantiation graph is used for the analysis and storage of simulation data. An example of a design instance graph is shown in Figure 2(c).

Listing 3: Code excerpt of the original testbench.

```
module B_tb(a, b, c);
  input  a, b;
  output c;

  initial begin
    a = 1'b0;
    b = 1'b0;
    #1
    ⋮
    a = 1'b1;
    b = 1'b1;
  end
end
```

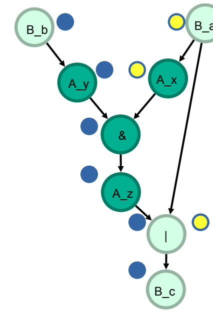


Figure 4: Visualization of IFT on the example circuit.

Design Choice 3 *Instation Graph Connections*

In the instantiation graph, nets are represented as point-to-point connections in the graph. Every net that connects multiple cells is represented by multiple edges in the graph.

3.2.2.5 Export Design The design can be exported from DuRTL back to a Verilog file from the main `Design` class (3). Additionally, the design is extended with logic for IFT based on Section 3.1.2. The port list of the module is extended with additional tag ports, which are added as additional wires in the module. The export function of each individual `Cell`-type also exports extra logic for IFT.

3.2.2.6 Exporting IFT Testbenches DuRTL utilizes existing testbenches (4) for the design to export testbenches extended with IFT logic (6). VCD data from the simulation of the original testbench with the design is used for the stimuli of the design inputs. This VCD data is parsed from a VCD file using a third-party library[4]. The IFT testbench provides the stimuli for the primary inputs of the design and the tag stimuli for the IFT tag ports for the simulation. The tags are determined by tagging heuristics (5). The tags are added to the testbench by extending the input and output ports of the testbench with additional tag ports. The testbench then supplies the tags as stimuli for the added tag ports of the design.

The tags are exported in one of 2 ways:

- in a text file that is read by the testbench

  Instead of writing the tags directly into the testbench, i.e., into an initial block, the tags are exported in a separate text file that the testbench reads. Exporting new tags by overwriting the text file is faster than exporting the whole testbench every time the tags change.

- the testbench contains a logic that dynamically determines the tags based on the simulation data

  Some design signals are used as trigger signals by the testbench to determine the tags for the primary inputs of the design.

3.2.2.7 Simulation DuRTL uses external tools, e.g., Icarus Verilog[5] for the simulation of the design (7). The exported design is simulated with the exported testbench, and the simulation results are stored in VCD files that can be parsed by DuRTL. For simulation, the exported design can be simulated using either the original testbench file or a testbench exported by DuRTL that is optionally extended for IFT. The results are written to VCD files which can be parsed by DuRTL.

Design Choice 4 *External Simulation Tool*

DuRTL uses external tools for simulation to benefit from their optimized implementation. Moreover, using an external tool also makes it easier to integrate the simulation of the design with other tools that are used in

[4] `https://github.com/gian21391/verilog-vcd-parser`
[5] Icarus Verilog, `https://github.com/steveicarus/iverilog`

the hardware design process. For example, the simulation results as Value Change Dump (VCD) files can be easily imported into other tools for further analysis or verification.

The instantiation graph is a flattened representation of the design that contains all the elements of the design and their connections. The simulation data is stored in the instantiation graph as a property of the edges/connections of the design.

Design Choice 5 *Simulation Data Storage*
Storing all simulation data in the instantiation graph allows for simplified access when analyzing simulation data and tags jointly.

3.2.2.8 Tagging Heuristics DuRTL uses tagging heuristics to determine the tags for the IFT logic. Currently, two different tagging schemes have been implemented in DuRTL. The first tagging scheme is based on the VCD data of a given testbench. For every timestep and every primary input of the design and testbench, the tags are determined either by tagging every primary input at every timestep (full resolution) or by randomly choosing a subset of the primary inputs and time steps (random). The second tagging scheme is a dynamic scheme in which the testbench tags the primary inputs of the design in response to some predefined trigger signals. The trigger signals are defined in the testbench and are used to determine the tags for the primary inputs of the design.

3.2.2.9 Analysis The design can be analyzed (8) in different ways, optionally using simulation data stored in a design instantiation graph. Two exemplary analysis heuristics are explained in the following.

Most & least used path – Given all tags after simulation, we identify the net with the most tags and extend it to a path from inputs to outputs by a depth-first search collecting the most tagged predecessors and successors to find the *most used path*. The same approach is applied for the *least used path*, collecting the least tagged predecessors and successors instead during the depth-first search.

Most & least toggled path – Toggling, i.e., value changes of signals, are commonly analyzed in hardware design. Similar to our *most used path* and *least used path*, the toggling metric computes the path with the most and least value changes over the simulation time.

Figure 4 shows an example of how tags might propagate through the design. Both inputs *B_a* and *B_b* are tagged, represented by the yellow and blue circles, respectively. The tags propagate through the design and determine the tags for the output *B_c*. In this example, only the blue tag propagates to the output *B_c*.

3.2.3 Evaluation

DuRTL has been used in two use cases so far. The first use case is the analysis of hardware designs using IFT for general design understanding and comprehension [SMF24]. Table 3 shows an overview of the hardware designs that were used and their sizes. The table also shows the lengths of the identified paths for the most and least used paths and the most and least toggled paths for the example designs. By exploring specific paths extracted by the tool, e.g., the most used path, a designer easily gathers insight into typical behavior and its implementation. The more simple toggling-based analysis yields less representative paths. Further details also on the interpretation can be found in [SMF24].

The second use case is an analysis of the effect of approximate circuits in hardware designs using error-responsive IFT [SMN+24]. Error-responsive IFT means that the errors produced by the approximate circuit were tagged, and their influence on the rest of the design was tracked by IFT. Table 4 shows a heatmap of a full-resolution IFT run that shows how many tags that were injected at a specific input port (left side) reached different output ports (top side). By this, the impact of certain inputs on the different outputs is immediately clear for a designer using approximate units. Further details and results can be found in [SMN+24].

3.2.4 Extensions and Student's Theses

We explored various side paths and design alternatives for our project with the help of students. For the Bachelor and Master students, this had the advantage of being exposed to a development project including a larger code base. For the project, we were able to focus on the main research tasks. Some of these theses may serve as the basis for future extensions and research directions.

We started exploring options for a 3D visualization [3, 12] as a way to ease comprehension and understanding and to create a more intuitive way to visualize the results of hardware IFT. Moreover, we started backing

Table 3: The numbers of cells and nets for the designs.

Design	#cells	#nets	#in	#out	#tags	25% tags most	25% tags least	100% tags most	100% tags least	toggle most	toggle least
SPI_Prim.	408	419	8	7	576	15	31	45	8	18	30
SPI_Sec.	41	47	8	2	288	19	13	14	8	15	13
y86	297	366	4	7	10,432	50	20	50	20	20	34
yadmc	847	900	9	10	10,208	21	59	21	58	57	5
aes_128	82	161	3	1	96	158	60	158	60	62	60
usb2uart	3,248	3,904	50	35	7,712	24	24	4	4	9	4
openMSP430	1,982	2,518	24	29	2,944	75	12	75	12	57	35

Table 4: *Apprx. FFT*, full resolution tagging

	add 1_1	add 1_2	sub 1_1	sub 1_2	add 2_1	add 2_2	sub 2_1	sub 2_2	qr	qi
add1_1	0	87	0	87	0	87	0	87	87	87
add1_2	0	0	0	0	0	0	0	0	87	0
sub1_1	87	87	87	87	0	87	0	87	87	87
sub1_2	0	87	0	87	0	0	0	0	87	0
add2_1	0	87	0	87	0	87	0	87	87	87
add2_2	0	0	0	0	0	0	0	0	0	87
sub2_1	0	87	0	87	87	87	87	87	87	87
sub2_2	0	0	0	0	0	87	0	87	0	87
rstn	19	25	19	25	19	26	19	26	25	26
ena	158	174	158	174	157	175	157	175	174	175
xr	174	174	174	174	166	174	166	174	174	174
xi	166	174	166	174	174	174	174	174	174	174

the testbench-driven but also testbench-dependent approach with a formal approach using satisfiability modulo theories to identify possible information flow [9]. The formal approach then provides precise information according to Eq. (1) and can compute all possible information flows. These two directions are the most likely to be included in future versions of DuRTL.

Replacing dedicated data structures with a generic graph database is possible in principle but expectedly leads to performance problems [2]. Directly oriented towards design understanding, static approaches to detect pipeline structures [6],[SRKF22] and to identify state machines [10] were considered.

3.3 Deviations from the Original Concept

The project was executed as initially planned. An extension beyond the original plans is the consideration of large language models that gained significantly in performance and in their capabilities to explain code. Thus, we studied whether code explanation and testbench creation can be supported [1, 7],[RF24]. While the current capabilities are not fully satisfying yet, the projection indicates that this might drastically change in the near future.

This suggests discontinuing the originally planned path of extending the current implementations towards more complex understanding tasks by dedicated analysis algorithms. Instead, we want to reuse and refocus our existing implementation toward security analysis and test generation.

3.4 Activities and Approaches to Quality-enhancing Measures through which the Validity or Verifiability of your Research Findings was Ensured

The open source release of the DuRTL [MSF24] includes a set of benchmark circuits together with their testbenches and licenses collected from the public domain. Moreover, the code base includes unit tests applied in continuous integration during development.

3.5 Handling of Research Data

The work in [SMF24] gives an overview of the resulting tool and provides results. For reproducibility of the results, we (1) open-sourced the tool, so it is available to other researchers [MSF24] and (2) we have a persistent snapshot [11] of the tool consistent with the results in the publication. The study in [RF24] also has a reproducibility package [8] even though the results may continuously change while the large language models under consideration are updated by their providers.

3.6 Summary

Overall, the initially planned project goals have been achieved, plus making the results available to the research community by open-sourcing the tool. Various extensions were explored during the project, including the most recent developments in large language models. For the actual implementation, follow-up usage in test generation and security analysis has already been started in other projects.

3.7 Bibliography

[1] Ahsan Bashir. Evaluation of LLMs for hardware design. Technical report, Hamburg University of Technology, 2024. Project Work.
[2] Malte Burmester. Analysis of hardware designs using graph databases. Technical report, Hamburg University of Technology, 2024. Bachelor Thesis.
[3] Ahmad Hawat. Exploring digital circuits in a 3d gaming engine. Technical report, Hamburg University of Technology, 2023. Bachelor Thesis.
[4] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. Hardware information flow tracking. *ACM Comput. Surv.*, 54(4), may 2021.
[5] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. Theoretical fundamentals of gate level information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(8):1128–1140, 2011.
[6] Lukas Limberg. An algorithmic approach to identify pipeline constructs in digital circuit designs. Technical report, Hamburg University of Technology, 2022. Bachelor Thesis.
[7] Abdur Rahman. Evaluating LLMs for hardware design validation. Technical report, Hamburg University of Technology, 2024. Project Work.
[8] Abdur Rahman and Görschwin Fey. Evaluating the performance of large language models for design validation – reproducibility package, July 2024.
[9] Rakshitha Kirugavalu Ramesh. Register transfer level information flow analysis using satisfiability modulo theories. Technical report, Hamburg University of Technology, 2024. Master Thesis.
[10] Bennet Sahlmann. Finite state machine detection and extraction. Technical report, Hamburg University of Technology, 2023. Bachelor Thesis.
[11] Lutz Schammer, Gianluca Martino, and Goerschwin Fey. Usage driven relevance analysis for IP cores – reproducibility package, July 2024. Version 1.0.0, `https://doi.org/10.5281/zenodo.12605973`.
[12] Alexander Schmidt. Verständliche, übersichtliche und räumliche Visualisierung von Hardware-Entwürfen. Technical report, Hamburg University of Technology, 2023. Bachelor Thesis.

4 Published Project Results

4.1 Publications with scientific quality assurance

[RF24] Abdur Rahman and Goerschwin Fey. Evaluating the performance of large language models for design validation. In *IEEE International System-on-Chip Conference (SOCC)*, 2024. accepted.

[SMF24] Lutz Schammer, Gianluca Martino, and Goerschwin Fey. Usage driven relevance analysis for IP cores. In *IEEE International System-on-Chip Conference (SOCC)*, 2024. accepted.

[SMN+24] Lutz Schammer, Gianluca Martino, Amir Najafi, Alberto Garcia-Ortiz, and Goerschwin Fey. Detailed insight into approximate circuits with error-responsive information flow tracking. In *Int'l Workshop on Boolean Problems (IWSBP)*, 2024. accepted.

[SRKF22] Lutz Schammer, Jan Runge, Paula Klimach, and Goerschwin Fey. Design understanding: Identifying instruction pipelines in hardware designs. In *International Conference on Circuits and Systems Technologies (MOCAST)*, 2022.

4.2 Other publications and published results

[MSF24] Gianluca Martino, Lutz Schammer, and Goerschwin Fey. DuRTL – a tool for design understanding of RTL code, 2024. `https://github.com/TUHH-IES/DuRTL`.